MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1118

April 1989

An Object-Oriented Software Reuse Tool

by

Michael D. Monegan

Abstract

The Object-oriented Reuse Tool (ORT) provides support for the reuse of object-oriented software by maintaining a library of reusable classes, recording information about reusability, allowing easy exploration of library entries, and providing an extensible approach to facilitating reuse in a software development environment.

ORT takes advantage of opportunities provided by the object-oriented style of programming that are unavailable in reuse schemes for conventional programming languages. It also records information useful in assessing the reusability of library entries and in relating these entries to each other.

In the early design phases of object-oriented development, ORT facilitates reuse by providing a flexible way to navigate the classes recorded in a library and the information associated with them. It thereby aids in the process of refining a design to maximally reuse existing classes.

The implementation of ORT is extensible. A collection of other useful tools has been identified. These tools can be directly implemented within the existing architecture and would compose the remainder of a practical system useful in increasing the amount of reuse in an object-oriented software development environment.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the Hewlett-Packard Company, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens Corporation, in part by the Microelectronics and Computer Technology Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-88-K-0487. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, neither expressed nor implied, of the sponsors.

Contents

Acknowledgments

Any praise for this work should first be directed to those who have helped me so greatly at Hewlett-Packard, at the MIT AI Laboratory, and in my personal life.

First, I would like to thank those at HP's Waltham Division for allowing me to do this work there and helping me along the way. Ted Schmuhl, Jack Ward, and Steve Fiedler offered great insight and direction, giving me a wonderful environment in which to work despite their many other product-oriented responsibilities. Many intellectual contributions were also made by Jack Harrington, Rob Seliger, Beth Farrell, and Wayne Lim. HP provided the most helpful and friendly librarian I have ever met in the form of Bridget Cullen – many thanks go to her for satiating my research interests. Further, technical expertise which aided my development efforts was provided by Charlie Nuzzolo, Cynthia Clement, Frank Richichi, John Marold, Chris Garrow, Lance Smith, and Sue Poliner. Further still, I am very grateful to Robert Cohn as well as Jim Psiakis, Marc Zeitlin, Kam Chin, John Cadigan, Howard Sumner, and Mousa Shaya for providing me many machine resources over the past months. Vicki Guerieri helped tremendously with some of the figures in this document.

I would also like to thank my thesis advisor, Dr. Richard Waters for his guidance in my research as well as the inspiration he gave to produce a document better than I had ever thought myself capable of generating. I would also like to thank Howard Reubenstein and Yang Meng Tan for reading and commenting on preliminary versions of this thesis.

Finally, I thank my future wife Nancy for continually encouraging me to achieve, my parents for supporting me throughout my educational endeavors, and my friends who have looked out for me when "crunch-time" came (then decided to invite its relatives and stay over for a few months).

I owe a great debt to all of you. Thanks!

Two of the most prevalent technologies involved in attempting to support software reuse, libraries and object-oriented programming, are combined by the Object-oriented Reuse Tool (ORT). ORT assists in managing a library of reusable software components developed under the object-oriented programming paradigm.

The software library approach, although already realized in many environments, warrants further investigation. This common approach can be improved upon with more sophisticated storage, search, and retrieval tools, which can automate much of the effort required to use a software library. Library-based programming offers little flexibility in expression, since the user is constrained to use only the entries in the library. However, much benefit is derived from its ability to suggest completed software works as building blocks for future work.

The object-oriented programming style is recognized as providing benefit in producing reusable software designs and implementations [33, 11]. This technology offers great flexibility in the expression of potentially reusable components, but offers little to assist reuse of such components.

Figure 1-1 illustrates these two technologies (software library and object-oriented con-

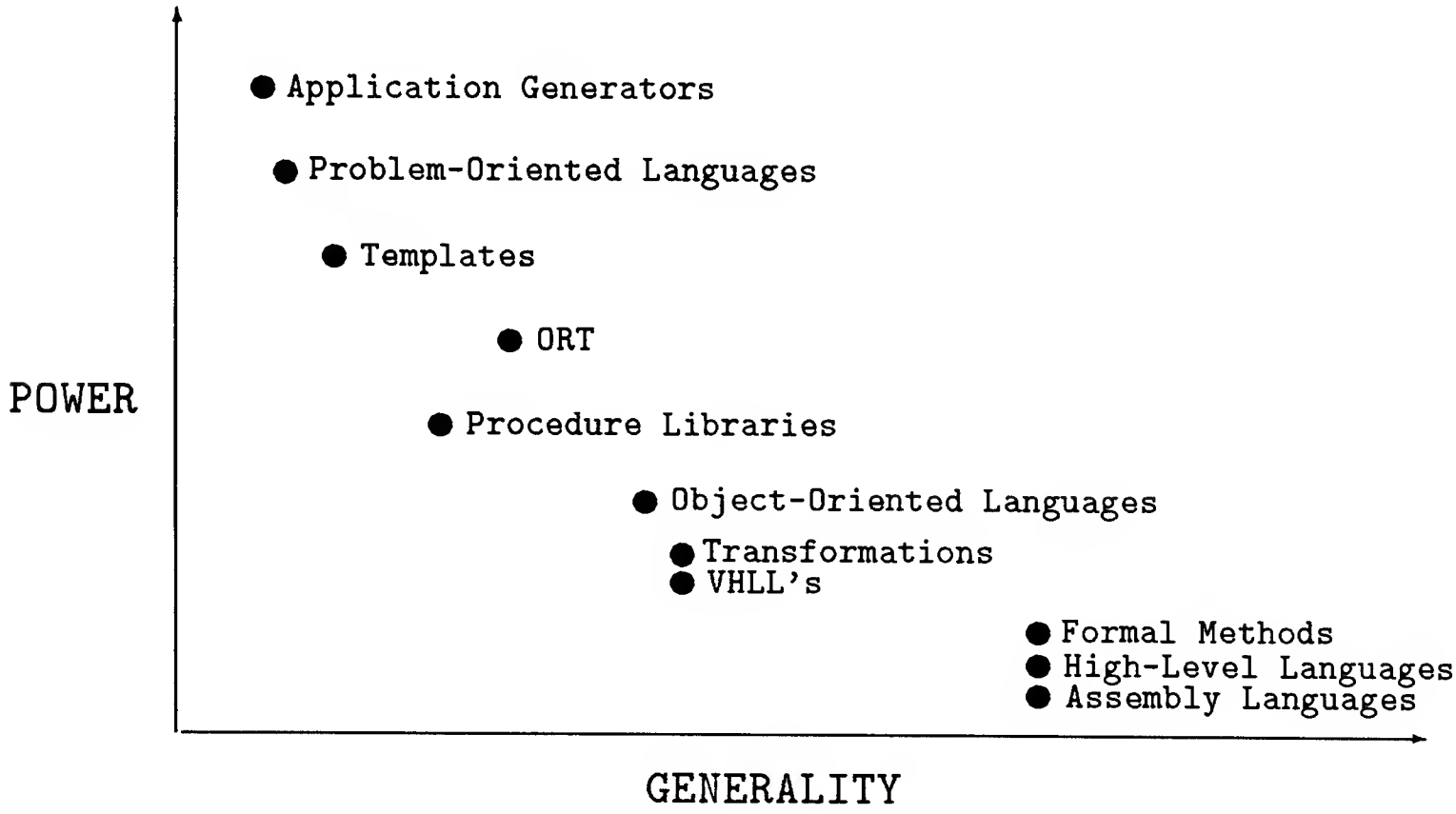

Figure 1-1: Characterization of reusability technologies.

struction) in the perspective of the many popular strategies used to attack the reusability problem. This diagram[1] roughly illustrates the spectrum that must be made in choosing between the currently available technologies: as reuse techniques become more powerful, they loose generality. For example, high-level languages and assembly languages are the most general forms of programming expression but offer little support for reuse. At the other end of the tradeoff, application generators are very powerful mechanisms for reusing the information needed to program in an application environment, but are so customized to the environment that they are not applicable outside their narrow domain.

An entry for ORT has been placed in Figure 1-1. Note that, relative to the traditional procedural library, ORT gains in both generality and power. ORT gains in generality because classes are more combinable than traditional procedural components, thus offering higher expressibility. It gains in power because of both the usefulness of the information ORT stores as well as the flexible interface ORT provides to access this information.

1.1 Software Libraries

Software engineers should, to be maximally effective, have access to a library of tested and debugged software components as well as information associated with their development and use. As more development occurs, such a library becomes increasingly effective as its base of entries grows.

Software library systems should not just be considered as tools to assist in the implementation of a system; they are design tools as well. For reuse to occur using a library-based technology, effort must be expended very early in the design process to insure that consideration is made of library entries that can be used in the implementation of such a design. One can not specify the design of an entire system top down and expect to reuse a significant amount of previous work. Rather, effort should be made from the beginning of the design process to insure maximal reuse. The design, therefore, must involve some bottom-up techniques based on existing components. Otherwise, any reuse that happens will occur only by accident or only at the lowest levels, where many programmers are taught to use the same abstractions.

Assuming that the quality of entries is kept at a high level by an ongoing editorial process, a library can improve a design by suggesting standard solutions to problems that have already been solved. If a design decision does not provide a better solution than does an entry in the library, then examination of that entry in the design process can implicitly suggest an improvement to this decision. This can result in standardization over all projects that use the library.

In addition to the direct reuse of library entries, limited modification and subsequent use of existing components, or *leveraging*, also occurs. The practice of leveraging, or "copy and edit", is well known to experienced programmers, because software reuse without

[1]This diagram is an adaptation of a similar diagram published by Ted Biggerstaff and Charles Richter [5]. It is not based on actual data, but rather on the observation that there is a basic dilemma between generality of application and payoff.

some modification rarely happens. A strong distinction is made by many organizations between strict reuse (i.e. no modification) and leveraging. This distinction is motivated by the risk that is taken in modifying any piece of software, since new defects are easily introduced.

However, the leveraging of code can, in many respects, be viewed as reuse of software at some level and thus any mechanism that supports unmodified reuse can also be used to support leveraging (i.e. if one reuses a component, it must first be found; if one leverages a component, it must also be found before modifications can be made). Additionally, the modifications required to suit the reuser, if done properly, may increase the generality of a component. This would produce a more valuable component to be entered into the library. For the purposes of this document, therefore, the distinction between reuse and leveraging is not significant.

The classification scheme used in a library system dramatically effects the retrieval process; it controls how like things are grouped together. A *faceted* classification method groups things by more flexible viewpoints or dimensions than does a traditional *enumerative* method (e.g., the Dewey decimal classification system) [34]. A *multi-faceted* classification system offers multiple dimensions along which a component can be selected, offering great flexibility, assuming that the facets and vocabulary terms are chosen with care. Such a system might generate many relevant entries on a given query, as well as many semi-relevant entries that, once examined, might contribute to the understanding of the system being designed.

1.2 Object-Oriented Development

In the object-oriented methodology of programming, an *object* is a self-contained entity with its own private data and a set of operations used to manipulate that data. Objects are instances, or particular values, of *classes*, which are, in one perspective, abstract data types. A class defines the data and set of operations for all objects of its type. It is therefore the basic unit of construction when writing object-oriented programs.

For the remainder of this document, the reusable software components being discussed are definitions of classes of objects and are usually referred to as "classes" (or "entries" when the statement applies to software library systems in general). Classes have associated with them *features*, a term used here in place of the collection of more language-dependent terms such as methods, instance variables, messages, member functions, data members, procedures, and the like. A feature is a generic piece of public knowledge associated with a class and can be a piece of data, a function, or a procedure. In this document, class names are words consisting only of letters, the first of which is upper-case (e.g., LinkedList), and features are words consisting only of letters, the first of which is lower-case (e.g., isEmpty).

ORT focuses on the object-oriented style of programming and the modes of reuse that derive from it. This is because the utilization of such a style increases the probability of producing reusable designs and code, which in turn increases the effectiveness of attempts

at collecting software components for future reuse.

Object-oriented programming helps produce reusable software, because it encourages data abstraction (providing modularity) and parameterization, desirable qualities for reusable components, and prevents the undesirable distribution of information throughout a program. Additionally, object-oriented programming provides a formal way to inherit features between classes. Inheritance is an efficient method of leveraging code while incurring significantly less risk of introducing additional defects than with the "copy and edit" technique. Risk is avoided because no modification of the original code is required and many details of the original implementation are hidden from the reuser. Finally, object-oriented programming allows for highly combinable components, but with loose coupling between them when combined. Because programming with objects is often more natural than with procedure or function calls, it is usually easier to express a system in terms of classes. Object-oriented languages afford the ability to bind a function definition at run-time based on an object's class. This allows loosely-coupled components, because objects need not know as much about each other.

In almost any object-oriented programming environment, a collection of often-used, basic classes are developed to support programming-in-the-small. These *generic classes* are reasonably standardized data-structures and are generally well understood by those with a computer science background. Typical examples are array, queue, and hash table. Generic classes are so well understood and becoming so widely available that they are now considered to be part of the programming environment itself (i.e. a necessary extension of the language).

An object-oriented programming style can be practiced in almost any programming language, but object-oriented programming languages more directly support this style than do conventional procedure- or function-based languages. There are many true object-oriented languages in use today, but there is no clear consensus on which will gain the most use. Because of this uncertainty and the fact that many class designs can be translated from one language to another with reasonable ease, ORT was built to be language independent, so as to remain flexible in this respect.

1.3 ORT

ORT combines two technologies, libraries and object-oriented programming, to provide library-based support for the object-oriented style of programming, in which a high reuse potential can be exploited. ORT manages interactions with a central repository of information about past software development activities. This repository forms a collective base of knowledge upon which a software engineer can draw to obtain classes used in previous efforts in object-oriented development.

Beside providing a library system focused on object-oriented components, ORT provides three additional major benefits:

1. ORT collects important information about reusability not usually stored in software libraries.

2. ORT allows a flexible way to navigate the information stored in the library.

3. ORT provides an extensible architecture on which to build other, related tools.

1.3.1 Information About Reusability

There are three basic roles information stored in the ORT library plays. The first role is as an indicator of how applicable an entry is to reuse. This information describes what the entry is, what it does, and how it is used. This type of information is common to most software libraries. The second role of this information, seldom realized in software libraries, describes relationships between entries. The third role of information in the ORT library, not usually realized anywhere, is to describe, regardless of how semantically appropriate an entry might be, its general quality and adaptability. The second and third types are considered to be new information with respect to conventional software library practice.

Aside from the information describing classes, their features, and their implementations, ORT keeps track of relationships between entries based on the dependency and inheritance hierarchies. If a potential reuser is interested in the details of a class, then he or she may also be interested in classes adjacent to it in either of these hierarchies. When reuse of this class occurs, many classes lower in the dependency and inheritance hierarchies are likely to be reused as well. ORT allows the user to examine these classes quickly, thus encouraging larger-grain reuse. If a class is interesting to the user, then classes based on this class (i.e. classes above this class in either hierarchy) may also be interesting, so traversals up the hierarchies can be made as well.

There are additional class relationships provided by the reuse process itself, because classes are developed based on library entries and subsequently added to the library as well. Examining these relationships allows the evolution of object-oriented software to be tracked. This is useful in assessing the benefit of domain analysis activities and in obtaining newer versions of abstractions quickly. All of these relationships provide the ability to move from class to class in the library in a nonlinear fashion, similar in manner to the browsing functionality in a hypertext system [10].

The data model of the ORT library also allows for the collection of information that can assist in the assessment of the characteristics relating to the reusability of the stored components. In particular, the history of successful and unsuccessful usage attempts on a component is recorded. Additional information used to classify, comment on, and formally review entries is stored in the library. Finally, code metrics (e.g., size, complexity) are recorded. All this data supports the activity of judging whether or not a given class is appropriate for reuse (regardless of its semantic applicability). This judgment is crucial to effective reuse and must be made quickly but correctly.

Aside from correctly deciding when *not* to reuse a component, it is also important to correctly decide *which* component to reuse. Faced with two or more slightly differing alternatives, the decision of which to reuse should often not be based solely on which

entry most closely approximates the desired entry, but also which entry has been shown to be more robust, adaptable, well documented, and, in general, reusable.

1.3.2 Navigation

ORT provides the ability to quickly locate information in the library. This capability is provided in two modes: querying and browsing. Querying allows the user to enter the library and quickly approach the most relevant entries. Queries specify a collection of relevant classes that are placed in an initial class browser. Browsing allows users to find classes they did not realize were interesting. Sometimes a problematic initial description of what is desired generates near misses that can be corrected by browsing along the links provided by class relationships.

Users can move from one collection of classes to others based on class relationships. Users can also browse off into reusability information associated with a particular class. Further, users can re-enter the query mode to refine the latest query and thus iterate the process.

1.3.3 Extensibility

ORT can be considered part of a framework of tools that is only partially implemented. To be practically useful in an actual object-oriented software development environment, a more complete set of tools, depicted in Figure 1-2, must be provided to help maintain the library and take full advantage of the information contained in it.

The library is implemented with a database that includes the information described above. Extensibility of this database definition is important, because the understanding of what information is useful to encourage effective reuse is likely to improve in the future. Along with the database, a set of database interface classes has been implemented (ORT itself is written in an object-oriented programming style), which interface the user applications to the database in an abstract manner.

Extensions to ORT, based on the current architecture, have also been designed. These extensions, described in Section 8.1, consist of tools not considered as interesting to prototype at this time. Such tools include a tool to automate entry of information embedded in code into the library, as well as some stand-alone tools that perform more specific functions or provide summaries of library contents. These tools compliment the current functionality of ORT and provide a framework which would include some way of editing the library.

Since library editing tools were considered appropriate to help in the prototyping of ORT itself, two additional tools, CHECKOUT and CHECKIN, were implemented as well. Beside affording high-level transactions with the library, these tools provide a locking mechanism to allow multiple users to simultaneously modify contents of the library.

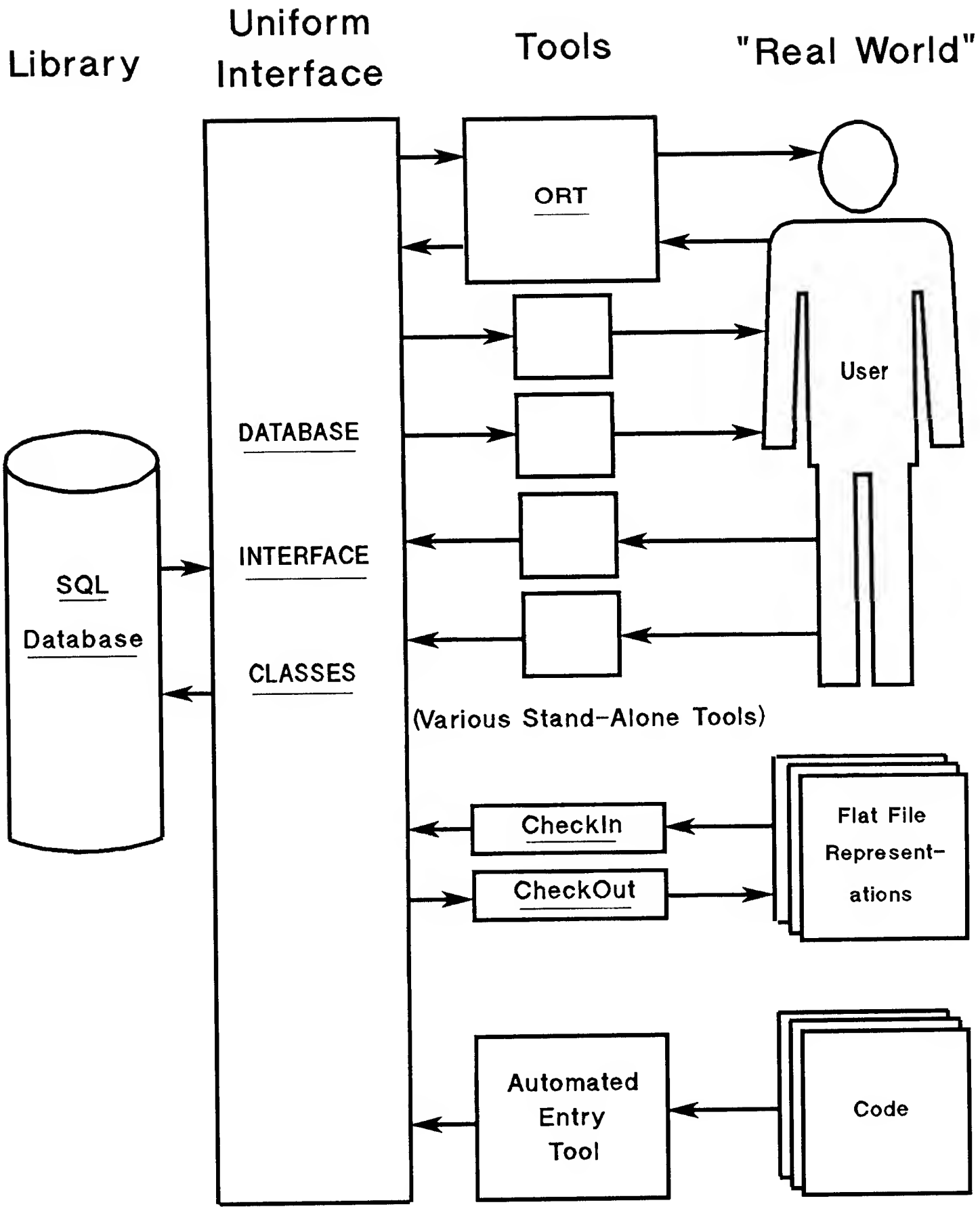

Figure 1-2: The eventual ORT framework of tools. Existing portions of this framework are denoted with underlining.

1.4 Roadmap

Chapter 2 states the problems plaguing software reuse, explaining why it is not more prevalent. An attempt is also made to counter each problem with a reasonable solution to suggest that, despite the many inhibitions to reuse, providing more tool support is a reasonable approach to improving the situation.

Chapter 3 describes in more detail the approach taken by ORT. It describes both the theory behind the design and the specific functionality the tools must provide in order to be effective. The approach breaks down into two basic steps: assessing reusability and finding relevant components.

Chapter 4 discusses the technique used to describe reusable components and depicts the *Class Description*, the basic unit of storage in the library.

Chapter 5 presents the design of the interface between the software engineer and the library.

Chapter 6 demonstrates how ORT is useful in an object-oriented software development environment by presenting an example scenario of its use in the refinement of a sample design.

Chapter 7 details the actual implementation of ORT. The technical details of the underlying database, the extensible architecture, and the non-obvious implementation issues are discussed.

Finally, Chapter 8, discusses possible further work and puts the work that has been completed in perspective. Limitations of the approach, logical extensions to the approach, alternative paths of exploration, and the applicability of ORT to software development methodologies that are not strictly object-oriented are discussed.

2. Problem Statement

It is well recognized that one of the most influential productivity factors in software development lies in the personal productivity of the software engineers doing the development [6]. A great deal of engineer-time is spent producing the custom software components needed to implement conventional designs. Still more time is expended on testing, debugging, documenting, and maintaining these new components. Clearly, minimizing this effort where possible would boost software productivity.

Since it is generally accepted that software reuse efforts can make a valuable contribution toward providing safe, verifiable, efficient, and reliable software that can be delivered in a timely manner [20], it is rather striking to note that the amount of code currently reused in industry is quite small. It is estimated that less than 5% of the code produced in an average software development environment is reused [13]. However, even ad hoc reuse of the software components generated in the process of completing projects may eventually provide a 25% increase in programmer productivity and would payoff the invested effort in two to five years [5]. In some cases, 40-60% of code was found to be repeated in more than one application [21] and reuse factors as high as 85% have been reported [20].

So as not to suggest that the problems implied in Chapter 1 are the only causes for lack of reuse, a more complete description of such causes is presented here. Given the great potential and such little actual benefit, it is important to identify the reasons why reuse is not more commonplace in today's industry. Certainly, the idea of reuse is not new; it has existed ever since there have been computers to program, dating back to 1950 with FORTRAN subroutine libraries. Additionally, published work on the subject appeared as early as 1967 with McIlroy's proposed software component's catalog [31]. Since then, limited-domain applications have shown excellent success [18, 29], but the general practice of software reuse has yet to pervade the industry and fulfill its potential.

There is not a great deal of reusable code hidden somehow in today's software development institutions. Rather, the problem is a basic lack of truly reusable designs and code. Although writing reusable software is quite time-consuming and costly, a methodology for writing and using reusable software must be encouraged if significant productivity gains are to be realized.

Opposing this challenge, many discouraging factors are collective cause for the lack of software reuse in industry today. Each of these problems is presented below along with a counter-argument, as best our current understanding can offer, to suggest that the problem is not insurmountable. The first two of these problems are, in fact, non-technical and thus not addressed by ORT at all. However, they are included for completeness; the solution to the problems ORT does address would be meaningless if the other problems were insurmountable.

2.1 Economics

The economic motivations of software production are probably the most influential inhibitors of software reuse. Simply put, much more resources are required to produce and maintain reusable software than to produce minimally functional software. There are several sub-problems:

1. **Producing reusable software is difficult.** In order to maintain the aggressive schedules motivated by commercial competition, software frequently gets released in a state that does not well serve future attempts to reuse it. Additionally, it never gets reworked later for the purposes of making it more reusable because no direct financial benefit is seen in this activity. These facts are seldom directly stated by those involved in software production, but with no financial incentive, why should anyone expect this investment of extra resources to be worthwhile?

   Solution: This phenomenon demonstrates the fact that there is still much progress to be made in the management of software projects; we are now only beginning to understand some of the issues involved. Specifically, if reuse was demonstrated to be economically effective, then its advantages in productivity would motivate the production of more reusable code. To be sure, reuse will become increasingly economically effective and commercially necessary as time progresses [26]. Therefore, this situation is not a cause for the lack of reuse, but due to a lack of vision for it.

2. **Commercial competition inhibits cooperation.** The copyright limitations, non-disclosure agreements, liabilities, and contractual considerations that result from commercial competition all inhibit the collaborations necessary for large-scale reuse. This is no small problem, as there are an increasing number of software development collaborations between industrial entities where third party efforts are incorporated into products to be sold. This form of reuse shows great hope, but introduces numerous other problems as well. For example, if a package of software components useful in the commercial or public domains must be withheld from distribution due to some proprietary information contained within, then the entire industry (save the originator) is effectively detained by an unproductive partitioning of software development knowledge.

   Solution: Software components should be viewed as a capital investment, a commodity that can be bought and sold much the same as are hardware components. Object-oriented programming makes this a particularly attractive business because:

   (a) Quality work can be advertised by distribution of the class interface specification, much the same as hardware IC's are advertised by their "data sheets."

   (b) Classes are usually standalone entities, encapsulated for semantic portability and information hiding, or part of a hierarchy disjoint from other sets of components

(c) Modification of the code for the class is usually not necessary in instances where inheritance can occur.

Eventually, the associated legal issues can hopefully be settled when it becomes more widely accepted that selling quality software components is a business like many others. Many facets of legal responsibility for software quality and safety are being identified. Eventually, the legal issues of support for purchased software components must be formalized just as is being done for finished software products.

Those involved in software development are often too quick to attempt to build from scratch something that is commercially available. The decision between buying and building should almost always be to buy because of the great costs hidden in development and, in particular, maintenance of software.

3. **The Economic Function of Software Development Firms is to Program.** It is currently believed that the value added by software development firms is in the art of providing a unique high-technology solution to their customers and that new, creative software will always be required. The limits on the domain of previously-produced code will always constrain the ability to reuse previous work in new software efforts. Thus, the art of programming is the function of a software engineer.

   Solution: As time passes, reusable code will cover more and more domains of programming. Therefore, the need for custom software will decrease with time. As stores of truly reusable classes are built and the practicality of object-oriented programming becomes more clear, a library-based tool can help minimize duplicated effort. This, of course, can only be done after a long-term investment has been made to assemble a critical mass of reusable library classes, an unavoidable cost in this approach to software reuse. Evolution of the "right" sets of intermediate abstractions will take time. As this occurs, the advantages of productivity will outweigh the "fun" of doing it all manually. In particular, it is expected that a shift in mentality will occur when software developers realize that the value they add is actually in the understanding of their user's problems and, on a high-level, how these problems should be solved.

2.2 There is No Reuse-Oriented Mindset

There are some prevailing attitudes that encourage programmers to "do it themselves." Even if reuse was easy, the reward system in today's industry encourages engineers to build from scratch rather than use someone else's work [37]. Additionally, experienced engineers sometimes become quite cynical of the concept of reusability because they have no doubt been "burned" in previous unsuccessful attempts to either create reusable software or reuse seemingly reusable software.

Programmers have most likely attempted to reuse software ever since programming was first practiced [31]. However, many of these efforts were not successful because our understanding of programming technology has changed so often. For example, years of work are cast aside each time a "better" programming language or environment is invented. Very seldom, if ever, has an innovative language definition been advertised as having a large base of pre-existing software upon which to build. Rather, language designers have either (1) forced a clean break to a new environment because no effective translation mechanism was delivered, or (2) provided downward compatibility that usually makes the language only a more complex version of an earlier one. Language designers envision people programming *in* a language rather than *into* a language.

Solution: These attitudinal trends, the Not-Invented-Here syndrome and the tendency to dispose of older programs due to (sometimes superficial) inadequacies, are major psychological foes of reuse, but are beginning to subside. There is an increasing awareness of reuse; those building software are more consciously striving towards, though usually not attaining, reusable constructions [24].

Changes of this nature usually come very slowly. Productivity measurements oriented toward production, not efficient delivery of quality code, impair the motivation for reuse as well. For example, managers are sometimes preoccupied with measures of code production per man-month. Essentially, this reward structure must be changed from the top levels of management down to the programmer.

Attitudes of both programmers and managers are hoped to be sufficiently mature and progressive so that this reuse inhibitor will have a decreasing effect in the future. These attitudes will change when it becomes obvious that they must in order to continue to improve our software productivity. Hopefully, the rising object-oriented programming trend, if it does nothing else, will help encourage the reuse mindset.

Software Manager's Reusability Theme

Ask not
how many lines of code did you write today?
but rather,
how few lines of code did you need to write today?

— Reid Smith

2.3 What Does "Reusability" Mean?

A formal definition of reusability does not exist. It would be hard enough to build reusable classes even if reusability was completely understood technically. However, the assessment of reusability is not widely understood, especially in cases where this requires

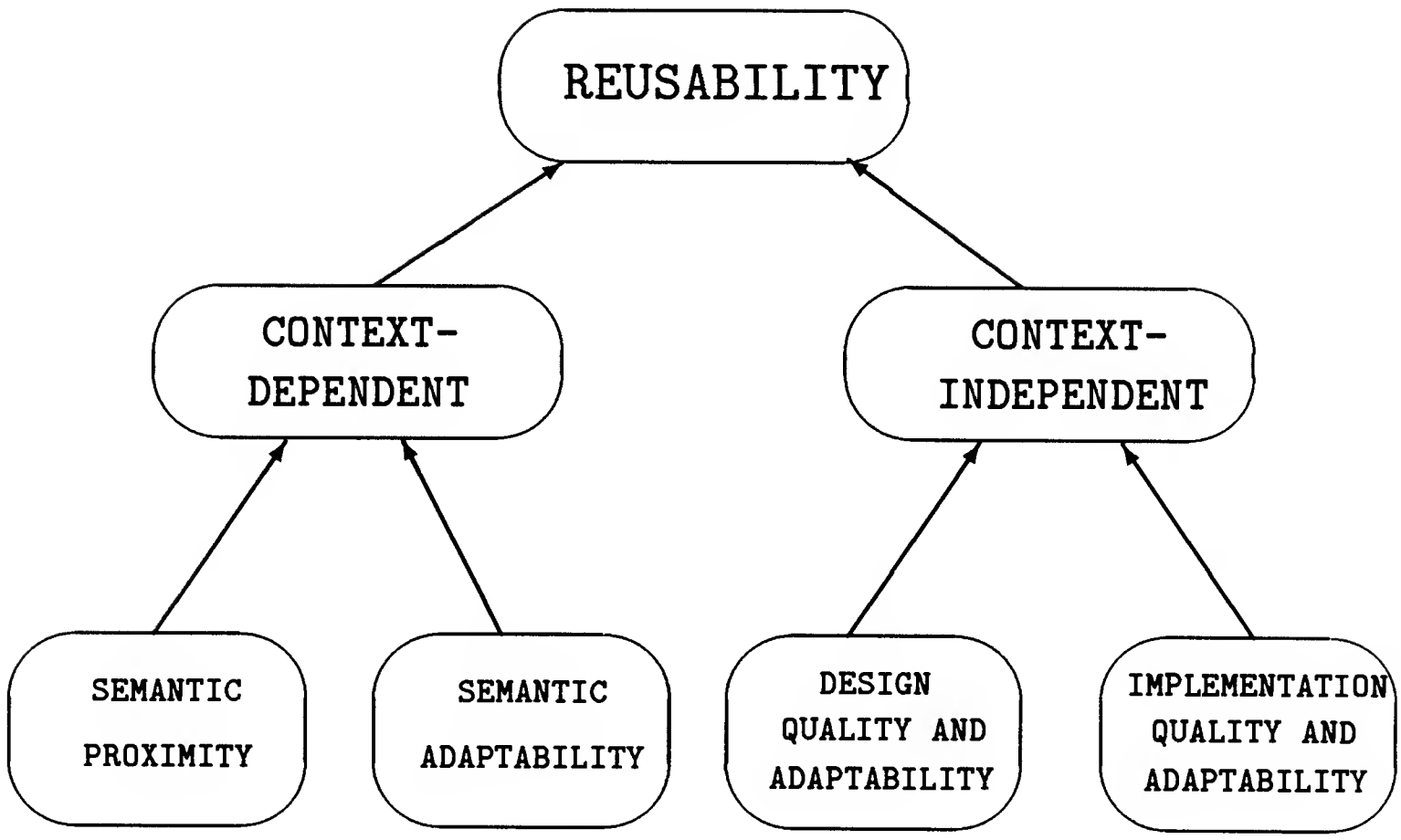

Figure 2-1: Decomposition of "Reusability"

a fair amount of domain-specific knowledge to make such a judgment. If managers strive for their programmers to produce reusable software, their programmers must understand what qualities are desired.

Solution: There is, as yet, no formal definition of the term "reusability." However, Figure 2-1 suggests an intuitive decomposition of its definition into the context-dependent and the context-independent.

Certainly, part of the definition is sensitive to the context in which you are attempting to reuse. Judgments of applicability (or usefulness) can only be made properly by the reuser at the time of the decision to reuse or not. It is therefore important to document the function of the library entries in preparation for the comparison to a potential reuser's needs.

The context-dependent reusability information can be further broken down into semantic proximity, or how "close" the entry is to what is actually desired, and semantic adaptability, or how easily the semantics allow the entry to be modified to what is desired. For example, a directed graph and a tree are similar semantically, and, because of their common foundation in graph theory, are also considered to be semantically adaptable with respect to one another.

A library should also contain entries believed to be widely applicable. These usually embody some useful piece of domain knowledge or at least display some conceptual value. Something outside of the implementation and design makes the class a generally useful concept to capture. For example, generic classes, by definition, are considered to fall into

this category.

Widely applicable classes can be identified by one of two methods, one formal and one informal. With the formal method, a library editorial committee would study existing programming and domain knowledge to identify useful classes for publication to a software production community. With an informal, survival-of-the-fittest method, ad hoc reuse would occur with little or no formal approval needed for entry into the library; anyone can add to the repository should they feel inclined. A history of reuse attempts is kept so that the most reusable entries can be identified and less useful classes can be pruned from the library. This type of maintenance process would then empirically yield widely applicable classes. Combinations of these two methods can also be employed.

The other part of the definition of reusability, as denoted in figure 2-1, has nothing to do with how applicable an entry is, but rather its *usability*. A class' usability is governed by its quality and the adaptability of both its design and implementation. Measurement and verification of design and implementation quality are current areas of research.

The context-independent reusability information can be broken up between design and implementation. For example, a component may be designed by a truly world class designer, but be implemented by a novice programmer. A reuser would like to be able to establish which aspects of the class are considered to adaptable and of high quality. Indications of the good design might be the intuitiveness of the documentation or a comment about the class' design by another programmer. Indications of poor implementation might be code metric measurements or poor test results.

As is discussed later, a proper solution involves both (1) establishing what information is relevant to each of the four "leaves" in figure 2-1 and (2) storing, maintaining, and presenting this information via tool support.

2.4 Reusable Designs are Rare

Software systems do not get reused. Rather, portions of systems get reused or leveraged. Unfortunately, no good representation for design exists that fosters reuse; techniques for generating reusable designs (designs containing reusable components, combinations of components, and sub-systems) are not well understood yet. However, it is at these high levels of granularity that the real payoff of reuse can occur. Therefore, reusability must be a deliberate and explicit design goal for any appreciable amount of code to be reused.

Much of the problem in creating reusable software is the lack of a way to properly write interface specifications. Object-oriented programming attempts to tackle this problem, by making programmers represent things abstractly and construct public interfaces to these abstract objects. However, a standard accepted methodology to specify the "correct" interfaces without a number of iterative trials does not exist.

This issue is further complicated in cases when no real embodiment of the design exists, save that which is embedded in code. For example, an algorithm developed in a prototype environment might be very useful in a production environment that uses a

different programming language. If the reusable part of the work, the design itself, is not explicitly documented, then the software engineer must "back out" the code into a general-purpose design before proceeding. This is a difficult task of "code evolution" with implementations not sensitive to issues of reusability and is even more difficult and error-prone than leveraging.

Solution: Progress is being made in the area of developing proper architectures that partition systems into reusable subsystems. Because this progress towards good system architectures is slow, alternatives should be considered. One alternative, a reverse approach, is to look at the types of parts that *are* reused from system to system, and see what types of systems can be built from them. Cataloging components for ad hoc reuse, especially from reuse-conscious designs, and collecting information on what actually did and did not get reused is a logical way to realize this approach.

Software engineers are, in general, becoming better accustomed to more standardized, clean abstraction barriers within systems so that changes can be partitioned and do not incur far-reaching side-effects. There is a growing movement to capture system-dependent details in implementations that satisfy a standard interface specification. An excellent example of this is the X Window System [17], which provides hardware- and network-transparent services to applications by appropriately-defined interfaces.

The challenge of building extensible and properly modularized abstractions is very real. As mentioned previously, this goal is more realizable in the development of object-oriented systems. Because the objects the system manipulates are generally more stable and better understood than the functions the system performs, object-oriented techniques generally provide more extensibility. However, only a relatively small number of systems implemented in an object-oriented technology have been commercially developed and are now ready to be extended or maintained, so little verification of this fact can be provided at this time.

2.5 Component Selection and Retrieval

Reusable components must be easily retrievable and selectable from the countless numbers that may eventually exist. For reuse to be effective, it must clearly require less effort to attempt reuse than to write the desired code. To reuse, one must (1) understand the code, (2) decide whether to actually reuse it or not, and (3) modify or inherit from it as needed. This mandates an effective library classification scheme and a good method of suggesting the characteristics of the desired component.

Solution: The retrieval and selection process has become better understood of late; numerous such schemes have been devised and shown to work and are now being evaluated over the longer term [34, 20]. More progress can be made, however, in helping to assess reusability once candidates for reuse have been identified. Higher rewards can be realized in the reuse of entire classes rather than simpler functions or subroutines.

2.6 Building Large Systems from Small Building Blocks

A fully parameterized set of reusable classes, if it existed and were used, could cause undue inefficiency and complexity. Construction of many layers of overly-generalized software, each of which performs only one small, discrete function, can result in a rather slow and sometimes overly-complex system. The classes can also impose their structure on the programmers using them, possibly limiting the programmers in expressing themselves.

Solution: This is really two related problems, one of efficiency in production and execution of software and one of controlling the complexity imposed on the programmer.

This efficiency problem can hopefully be rectified by advances in object-oriented language compilers that can bind together as much object code as possible at compile-time and heavily optimize the result, while allowing small, incremental changes to be easily made. Additionally, hardware price/performance ratios continue to drop. Computer scientists traditionally sacrifice frugal resource utilization for complexity management and productivity enhancements, just as hardware engineers often use standard parts at the expense of "wasting" circuitry in digital design. There may be limits to the practicality this philosophy, but for now it is a true fact of life.

The real problem is understanding all of the small pieces of a system and how they interact. Tool support can play a large role in assisting the programmer with the management of this information. The imposition of the structure of the class collection has already begun. In fact, the increasingly-popular object-oriented methodology is intended to impose a new data-directed style of thinking upon the programmer. In the focused view of software reuse discussed by this document, this imposition is a necessary part of proper (in the object-oriented sense of the word) programming, because it centers the organization of the system around a more natural representation of "real-world" objects.

2.7 Summary

The conclusion to be drawn from this exposition on inhibitors of reuse is that, although there may be some non-technical problems at work that contribute to the lack of software reuse, there are also problems that *can* be solved with technical progress. To summarize, the issues discussed can be grouped under two major meta-problems:

1. The lack of motivation and limited ability to produce truly reusable code.

2. The need to facilitate reuse, assuming reusable code exists.

Encouragement of reusable software production, usually considered a managerial problem, requires technical support to clearly define what reusability means, so that programmers have a more concrete methodology and a set of standards in mind when encouraged to produce it. Clear benefits of reuse must be demonstrated to motivate the managers as well as the programmers. Programmers should be encouraged to prepare

an assessment of reusability upon submittal of their classes to the library forcing them to think about the reusability of their code and the ramifications of their decisions upon potential reusers of it. Programmers should also be equipped with the proper tools to produce more reusable software.

Facilitation of reuse, as mentioned previously, is now beginning to be accomplished through the use of interactive tools such as ORT. However, further progress can be made, especially in conjunction with object-oriented programming. Potential improvements can provide a more sophisticated selection and retrieval process, discussed in later chapters.

3. Approach

The types of reuse ORT targets are specific not only in nature (object-oriented), but in granularity as well. Aside from the reuse of personnel with the appropriate knowledge and experience to increase software productivity, there appear to be four levels of reuse in software development:

1. **Programming Knowledge Reuse**

   The reuse of ideas, concepts, knowledge and programming clichés represent the lowest level of reuse. Techniques that codify the activities of a programmer, attempting to duplicate human problem-solving [35], can boast a very large, long-term payoff, but much work still remains before a useful "automated programmer" can be realized.

2. **Low-Level Component Reuse**

   The reuse of individual software components (i.e. the leaves of a module-dependency diagram) represents a more common level of reuse. These components are sometimes readily reusable because they are common between projects, generic in function to the domain of interest, and easy to understand and use. Reuse already occurs, to some extent, at this level because programmers understand this type of reuse well and the need for its management is minimal. It can, of course, be encouraged and facilitated by a library-based tool, but nearly any simple repository (e.g., a file cabinet of documentation and code listings with index tabs) might suffice. The payoff of this type of reuse quickly reaches a limit difficult to exceed, caused largely by (1) designs that dictate custom components rather than utilizing existing ones and (2) top-down designs which yield tall dependency hierarchies of components, whose leaves are highly context-dependent.

3. **Aggregated Reuse**

   The reuse of intermediate level objects and aggregate collections of interrelated objects appears to be the most interesting level of reuse. At this level, the best compromise between commonality and reward-level occurs. Although reuse occurs to some degree at this level, it is severely hampered by many issues discussed in the previous chapter. Aggregated reuse shows the most immediate promise of improvement; ORT is therefore proposed to encourage and facilitate reuse here. Although reuse at these higher-levels is not as well understood, the benefit in automation is much greater here because the complexity of object relationships must be managed. This complexity is not found at lower levels of reuse. The classes can be presented in such a way so that they may be understood quickly by

a potential reuser, who may wish to reuse existing collections of objects, or combine existing objects in new ways.

4. **Design Reuse**

   The reuse of subsystem and package designs is the most powerful level of software reuse. At this level, it is the high level interfaces that are important to the reuser, not how they are supported. Design reuse provides a much richer approach to building systems and collections of systems effectively than do current design techniques. Little reuse can occur between dissimilar projects, but the rewards are high in the cases where it is possible. To foster such cases, design-for-reuse efforts must determine how to provide a common base on which many systems can be easily built. Theses efforts typically result in the embodiment of domain knowledge in a platform of software design, implementation, and documentation.

3.1 Realizing Aggregated Reuse with ORT

Those involved in software development have long hoped to realize a process by which non-trivial software components could be produced and effectively reused in future software products. The two main problems involved in developing such a process are:

1. How do you make software components more reusable?

2. How do you maximally reuse these components in subsequent software development?

3.1.1 How ORT Helps

In ORT, the first problem is approached by recording information about reusability that is useful to the user during the component selection process. This provides a platform on which those using ORT can build a better working definition of reusability. This forces the reusability of a class to be considered "up front" so that changes can be made to improve its reuse potential before it becomes a stable library entry. The description of this information about reusability is the subject of the following chapter.

The second problem is approached by assisting in the retrieval of software components relevant to a design and providing easy access to the relationships between components in the library (e.g., classes that have previously been used in conjunction with an "interesting" class). Assuming that there are relevant classes in the library to find, the process of finding them can be expedited by the use of ORT. This is the subject of Chapter 5.

3.1.2 How ORT Does Not Help

It is also useful to state where ORT does *not* attempt to assist in solving the two above problems.

- **Producing and assuring high-quality entries.**

  ORT does not perform software synthesis nor does it critique class designs and implementations. Rather, it is a tool that can be used by those who care enough about software reusability to produce reusable classes and have an editorial process (most likely a review committee) to monitor the quality of an ORT library. In such an environment, ORT can remove some of the drudgery associated with software reuse and enhance the process with automation and mass-storage management.

- **Making library entries easy to configure together.**

  Although programming in the object-oriented model assists in this endeavor, ORT makes no attempt to help make its entries more uniform or configurable with respect to one another. It would be appropriate to continually ensure that library entries are kept consistent and easily-combinable.

- **Continually improving library entries.**

  Although ORT affords the ability to find components and record improvements and enhancements of them, it can not actually improve the components in its library.

3.2 Assessing the Reusability of Classes

In constructing a library filled with reusable code, some effort must be made to insure that its entries are of sufficient quality (or at least known quality). Potential reusers will peruse the contents of this "software component warehouse" in search of a reasonable solution to the current problem, and, therefore, the quality of relevant classes is of paramount interest to them.

The information recorded by ORT is based upon the model presented in Figure 2-1. The particular information chosen to be recorded in the library is the subject of the following chapter.

Further complexity is added to the process of reusability assessment by the fact that the definitions of some characteristics are not universal to all potential reusers. "Fuzzy" terms should be customized to the individual's level of experience, knowledge of the application domain, and personal opinions [34]. For example, a novice programmer might consider a certain program to be much "larger" than would an experienced programmer and have higher expectations for the level of documentation required to properly submit it for general reuse. As will be shown in Chapter 5, users are given a large amount of control over how querying this information is accomplished.

3.3 Retrieving Software Modules

A potential reuser would like to build a system he or she does not completely understand yet and would, as quickly as possible, like to build this from parts that may or may not exist. In short, finding appropriate reusable components is difficult.

ORT's method of selection and retrieval has two phases: querying and browsing. The first phase consists of developing a query that describes components of interest. In this phase, it is important that the user formalize exactly what is desired. The result of the querying phase gives the browsing process somewhere to begin other than at the highest level of abstraction over the library contents.

In the browsing phase, truly relevant candidates for reuse can be uncovered. It is unlikely that effective browsing should begin with a collection of all the classes in the library. Rather, the browse should be focused as quickly as possible. The method of focusing the browse must be flexible. Browsing in separate "domain spaces" has been demonstrated [2]. However, the categorization of classes is inherently flawed because (1) classifications can not be universally agreed upon and (2) categories evolve as programming domains are explored [34]. The querying phase implemented by ORT offers a much more flexible notion of partitioning the contents of the library into interesting possibilities and uninteresting ones.

Once ORT executes a query, a set of matching entries can be examined using ORT Browsers. ORT Browsers are autonomous X-Window user-interface objects that allow the user to examine a collection of entries and execute commands relative to them. The initial ORT Browser generated by the querying phase contains classes from the library that match the query. ORT Browsers can generate other ORT Browsers, allowing users to follow relationships like links in a hypertext database. For example, if a user is interested in the classes upon which an interesting class depends, he or she would select the class of interest, then hit the "DEPENDS ON" key to produce another Browser Widget containing all the classes in the library upon which the implementation of that class depends.

A more complete description of querying and browsing with ORT is contained in Chapter 5 and a demonstration is presented in Chapter 6.

4. Describing a Reusable Class

The most critical design decision of ORT is the choice of the data contained in the library. The model of this data crucially determines the library's ability to store and organize the many different types of information about software development activities. This chapter presents a number of different views of this data.

A wide range of approaches can be considered in organizing this data. At one end of the spectrum, the library can be merely a collection of files and directories, possibly distributed across machines, but made to appear centralized with commonplace networking tools. This technique is easily implemented and, in fact, already employed at places in industry where attempts to collect useful software components have been made. However, this technique lacks a mechanism to realize relationships between the various pieces of information, save for the large-grain organization of file naming and even higher grain directory structure.

At the other end of the spectrum, a full-fledged hypertext representation offers great benefit as a very general solution. It can provide the ability to link related pieces of information, browse across the library's information in an unstructured manner, provide an effective interface to the user, and do so with a high degree of versatility. However, this solution also has problems: software engineers would be faced with a learning curve to use hypertext as well as a training curve, because without structure imposed upon the use of such a system, it would quickly become ineffective due to lack of organization.

The approach selected in representing the data lies between these two extremes. It captures the flavor of the hypertext solution in that (1) data elements are considered objects that have relationships to other types of objects and (2) there are links that connect pieces of information together. However, the ORT implementation is based upon a much more stable and well understood technology.

The ORT library is highly structured to promote a single, clear view of its contents. This structure is imposed by having a few well-defined types of objects and links rather than the conventional, less-structured hypertext organization. Object-oriented programming and software reuse are still active research topics. Therefore, the structure, though static for a given instantiation of ORT, has the ability to be easily changed, meaning that the understanding of the data can evolve over time. The way in which this is accomplished will be discussed in later chapters.

4.1 An Initial View of Class Descriptions

Information about a class in the ORT library is organized into a structure called the Class Description. The Class Description is the main type of element in the library and represents a class that can be reused in some way. The Class Description can be viewed in

terms of the model of reusability presented earlier (see Figure 2-1), where all the information relevant to the reusability of a library component can be grouped in four categories: semantic proximity, semantic adaptability, design quality, and implementation quality. Each of these four areas are addressed by the structure of the Class Description, as is described below.

4.1.1 Semantic Proximity

The only reliable technique available to assess semantic proximity is to allow the user to compare Class Descriptions in the library with conceptual ideas about what is desired. Automation of this process is much too ambitious a goal for this project. However, assistance can be provided in this effort by augmenting the user's abilities with a storage and presentation tool. To do this, results of software development must be described in the library in such a way as to promote the comparisons that must be made. This is not a radical concept, since any library must, at the minimum, be able to describe its contents.

The results of object-oriented software development can be organized into six major categories, as shown below:

More Difficult/Lower Yield:	Less Difficult/Higher Yield:
• Requirements Documentation	• Descriptions of Class Behavior
• System Design Documentation	• Class Interface Specifications
	• Class Implementations (i.e. code)
	• Test Suites (i.e. plans and code)

The categories of reusable information on the left (above) are not incorporated into the Class Description because of both the difficulty involved as well as the lack of obvious benefit. They are difficult because requirements and system design documents often include diagrams, report layouts, and screen designs. Such information is not only demanding to store in a library, but even harder to select and retrieve in a reasonable way. These two categories are also considered to be, in general, much less structured collections of information with a reuse potential heavily restricted by their level of similarity between projects. Therefore, ORT does not attempt to facilitate reuse of this type of information.

The other four categories of reusable information can be considered to be associated with at least one class. In some cases, the information is shared between classes (e.g., portions of class interface specifications are shared between classes that are related through inheritance). Therefore, these four types of information are incorporated into the Class Description. The description of the class is stored in three forms: the full textual description of the class, a shorter description limited to 100 characters or less which summarizes the role the class plays in a software system, and a collection of keywords that can be

used for retrievals based on the class description. The description of the class interface specification is stored by linking a class to a set of features, described later. The code and test suites are linked to a class by storing file names in the library, allowing access to the class implementation (and verification thereof) from within ORT.

4.1.2 Semantic Adaptability

Information used in judging semantic adaptability overlaps, to a degree, with that used in judging semantic proximity. However, the user is relied upon even more heavily because he or she must provide a perspective on the semantic "terrain" between that which is desired and classes contained in the library.

A truly reusable class should be capable of adapting to express many semantically differing classes. Save for two pieces of information, however, little information can be stored in a Class Description to suggest the degree to which this is true for the class it represents. First, comments about the class are stored; the author or authors of the class are, in theory, the best authorities on what the class can be expected to represent. Thus, a comment is solicited from them upon submittal of the Class Description. The views of those who designed and implemented the class may be useful in judging semantic adaptability when browsing the library. Second, class invariants are stored in a field separate from the description of the class. Invariants are important because they describe conditions which are always true with respect to the class representation. When making a semantic translation of a class design, a shift in constraints may occur. The invariants of a class suggest what these constraints are and allow the ORT user to observe whether or not these constraints conflict with the new application of the class.

4.1.3 Design Quality and Adaptability

Because the ORT library contains only object-oriented software components, worries about side-effects, encapsulation, and modularity are minimized. However, the user must be assured of design quality quickly or discouraged from reuse before too much investment is made in investigating a class.

Information that helps in this assessment serves two purposes: that which demonstrates that the design is well thought-out and that which helps to convey the class interface (i.e. how to use the class). First, some indication should be provided to suggest that the class has a formal basis and has been verified to work. The class description and invariants help in this effort, as do comments from those performing quality assurance tests. Additionally, the relationships between classes, as described earlier, help convey a larger design picture which the potential reuser can evaluate. Second, feature descriptions, if well written, can convey to the user how the class is used, because the interface is the only way the programmer interacts with a class. Thus, the features exhaustively document, using names, parameters, and descriptive text, the use of the class.

4.1.4 Implementation Quality and Adaptability

The assessment of the class implementation is assisted in three ways. First, a direct assessment of the class implementation can be made be examination of the code, allowing the potential reuser to examine the actual modifiability of the source. Second, summaries concerning the state of verification can assist in the assessment of implementation quality. Finally, measurements of code metrics can also suggest potential problems, should they exist, in the implementation.

4.2 The Contents of Class Descriptions

This section contains a more complete view of the Class Description. An entity-attribute-relationship model of the Class Description is depicted in Figure 4-1. An accompanying textual description, centered around the entities in the figure, follows:

1. **Class** – The nucleus of information in the Class Description contains basic, central information about the class itself:

   - Class name.
   - Alternate class names (aliases).
   - Class revision – compatible with external version control systems, if used to manage the actual code files.
   - Revision time – the time the above revision was made to the implementation.
   - Submitter – the name of the person who last submitted the Class Description to the library.
   - Checked out – a lock to inhibit simultaneous edits on the Class Description. This attribute contains the user name of the person applying the lock, otherwise this attribute is empty.
   - Checked out time – valid only if the previous field is non–empty.
   - Short description – A one-line summary description of the role the class is intended to play in a software system.
   - Long description – The class design documentation of unlimited length, describing the role and overall behavior of the class.
   - Code metrics – Metrics associated with implementations of classes give some numerical estimations of intuitive attributes that are perceived to be graduated across a spectrum. These are often based on code size, measured in Non-Commented Source Statements (NCSS) and consist of:

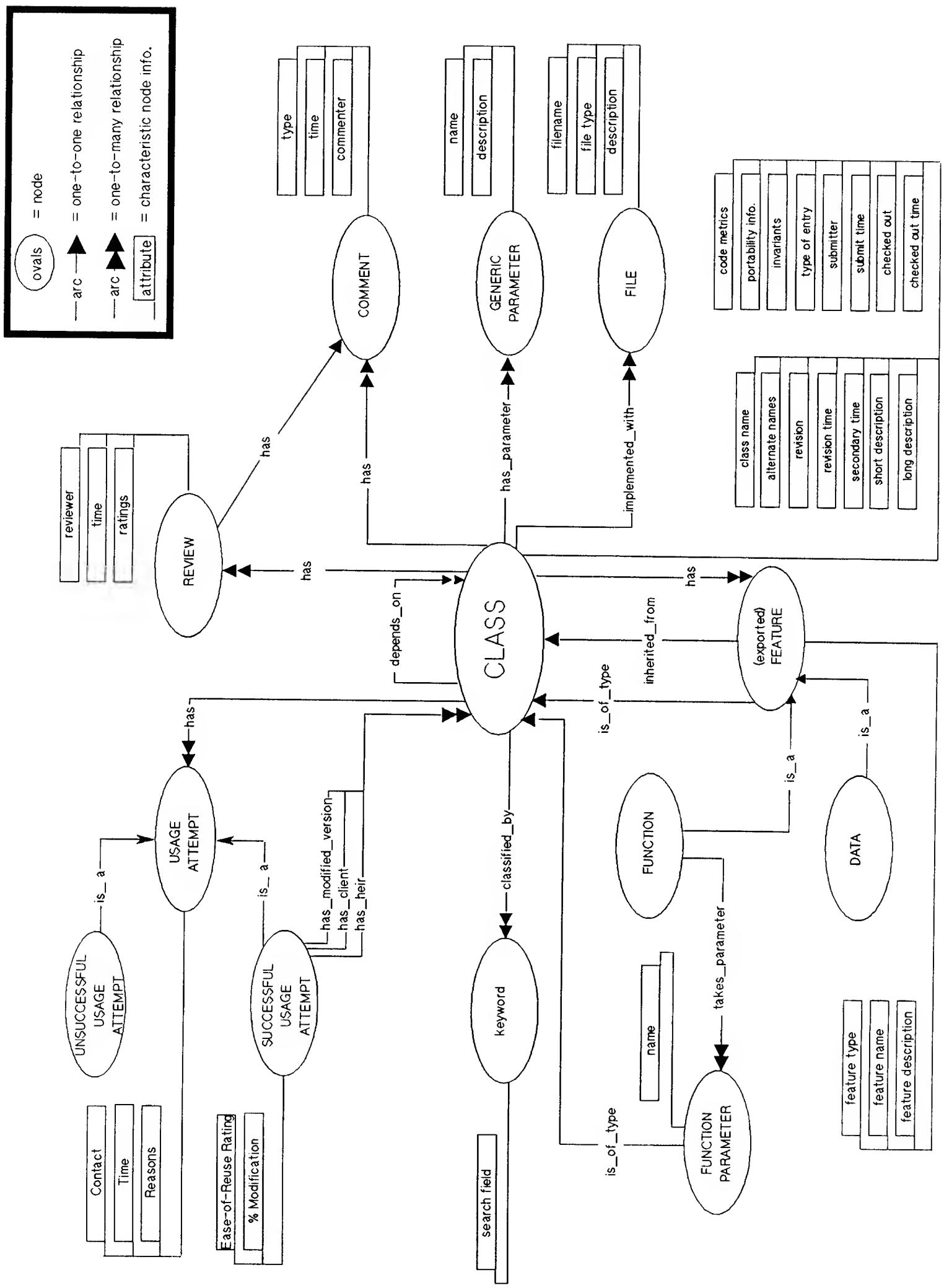

Figure 4-1: Entity-Attribute-Relationship Model of the Class Description

- size (NCSS)
- # outstanding defects
- # defect repairs
- # outstanding enhancement requests
- size of test code (NCSS)
- McCabe "composite" complexity
- Defect Density (#/NCSS)
- Mean-Time-To-Repair (days)
- # lines of documentation/NCSS
- # functions/NCSS

- portability information – A description of the environment needed to take advantage of any code associated with this class (i.e. language, compiler, Machine, OS)
- invariants – Textual information describing semantic invariants of the class. These could be considered part of the long description, but are separated to focus special attention on them, as they can be beneficial in asserting the correctness of a class implementation. This text could contain prepositions meaningful to some external specification tool.
- type of entry – The type of class description entry this is:

  U: unsubmitted – automated data-entry tools would produce this type of entry, which could be converted to type S via an interactive approval process.

  S: submitted

2. **"Depends_On" Relationship** – This binary relationship connects pairs of classes and makes up the dependency hierarchy of modules within an object-oriented software system. A Class Description documents a class specification and an accompanying implementation. Should this implementation be updated, causing the class revision to be updated, a new, different Class Description is created. Therefore, a class statically depends on any number of other class definitions, which, ideally, have corresponding Class Descriptions entered into the library as well. A class A depends on another class B if an object of type B appears in the specification or implementation of those features defined by A.

3. **Features** – All data, procedural, and functional aspects of a class are considered class features. Essentially, this is the description of the public interface, or protocol, of the class. Features are categorized into types, which are themselves defined in the library. Though feature types may evolve as our understanding of object-oriented programming increases, the current break-down exists as follows.

   (a) Data

   (b) Constructor

   (c) Destructor

   (d) Mutator

(e) Observer

(f) Observing Constructor

(g) Observing Mutator

(h) Other

The distinction between a feature implemented as data and as a function is subtle. The main difference is that functions take parameters, which have an associated name and type. Therefore, the function and function parameter entities are added to the model to appropriately record this information. A data entity takes no such parameters. Each feature is defined in one class (a class "has" a feature). The hierarchy of inheritances is recorded on a per-feature basis so as to not exclude languages in which classes can selectively export inherited features (e.g., Eiffel [33]). Each feature has a name and a description that serves as documentation. All features have a type, including procedures that return void. This data-type or return value (depending on the feature type), if represented with a Class Description currently in the library, can be accessed via the "is_of_type" relation.

4. **Usage History** – Information concerning the history of the use and reuse of a component can be quite valuable for a number of reasons. First, if a number of software engineers have chosen to use a class, then it must be worth at least considering. Second, if an engineer considers using a class and finds some serious deficiency, he or she should be able to summarize these negative findings in the library to save the same time of consideration that might be taken by others. Should this deficiency be corrected in another version of the class, a reference can be made to another Class Description in the library. Regardless of any other information that can suggest qualities relating to reusability, this history can empirically demonstrate reusability (or lack thereof). Third, this usage history can provide the names of programmers that should be contacted in the event of an update due to a "bug-fix" or enhancement. This provides an incentive to register reuse efforts in the library, as this will allow eventual notification of relevant changes. Fourth, The usage history serves as a collection of examples on how the class is used, which can be important for programmers who learn well by example.

   The basic unit of history is a usage attempt, which can be successful or unsuccessful. A successful usage attempt has some attributes associated with it. These include the name of the reuser, the date, a subjective rating on the ease of reuse, how much modification was required and why it was needed, a set of clients that utilize the class in that specific application, and a set of heirs that inherit properties of the class in that application.

   An unsuccessful usage attempt documents potential reasons why someone might *not* want to use a class. Sometimes engineers are "burned" when class implementations initially seem useful and reasonably robust, but are found to be not so after much

wasted effort. To avoid this, it is encouraged that anyone checking out information about a class later record whether the class was useful or not (an electronic-mail reminder system could be implemented as a periodic system task (e.g., with cron(1)) that would send out messages to those who have not yet reported what actually happened, a reply to which could be parsed and stored in the library).

A major problem with this approach is the assumption that it makes about the enforcement of appropriate levels of documentation. In a sense, it becomes the software engineer's *duty* to report his or her activities to the ORT library, so that others do not waste time figuring out things that have been already analyzed. As with many approaches to reuse, apathy is the worst enemy. If ORT is ignored by some, its utility to others is reduced. Direction from management must encourage the extra effort to make reuse happen.

5. **Classification and Indexing Information** – The classification scheme for class descriptions is important in accessing collections of classes grouped semantically in some way. This is accomplished by structured keyword accessing. Associated with each keyword is a search field. This search field indicates the location in the underlying database schema where the keyword was found. The search fields are themselves stored in a special table in the database to afford flexibility as the model evolves.

   In addition to the finer-grain classification of keyword-indexing, a more general categorization can be made by storing category names as special types of keywords. These categories can be instantiated as the collection grows and needs reorganization. The motivation for categories comes from broader "What's out there?" kind of queries. For example, a user might ask to see everything under the category "Graphics" and receive, among others, a Class Description for class Menu, which might also have a keyword entry for the category "User Interface."

6. **Files** – Files used to implement and verify a class are also recorded in the Class Description. Any number of file entities can be associated with a class. Each file has associated with it a file-system-based path making it accessible to ORT. These files have, as attributes, this path, a description of the file's contents, and a file type. Like the feature types described earlier, the file types are themselves stored in the library to allow future evolution. In the current instantiation of the tool, the following file types are pre-loaded as a basis for testing using classes implemented in C++.

   - Specification/Include File
   - Source Code File
   - Build Script/Makefile
   - Documentation File

- Test Plan
- Test Code
- Test Results
- Defect Report
- Enhancement Request Report
- Unknown File Type

Access to code is of paramount importance to the reusers who are also the programmers responsible for integrating classes chosen from the library into a new software system. Completed test plans and results give the reusers an idea of the level of robustness they are assured. Also, test-related metrics can be computed (i.e. test NCSS and class NCSS) that have been shown to correlate to defect population and complexity [19]. The archiving of test plans assists in the testing of future similar classes or in the testing of modified versions of the archived class.

7. **Reviews and Comments** – The subjective opinion of software engineers concerning classes they have studied is considered valuable information to store in the library. It allows a "discussion" to form that is focused on the issues of making specific classes more reusable.

   This type of information manifests itself in reviews, consisting of a set of ratings by a reviewer, and comments, which are textual pieces of information that can be attached to a Class Description. The types of comments capable of being entered are, in general, very loosely structured and can evolve as the system becomes better understood.

   Subjective quality opinions (rating of 1 to 10) can be recorded from those software engineers deemed responsible for reviewing a class and its associated Class Description entry. There are a number of relevant ratings:

   - specificity
   - functionality documentation quality
   - interface to other abstraction levels
   - separation between code and specification
   - generality of class
   - overall quality
   - documentation on how-to-use
   - support for debugging
   - general readability of code
   - clarity of a semantic role

   As mentioned earlier, it is also appropriate for the author to include an assessment of portability, with particular attention paid to factors that make this semantic idea, design, or implementation *not* reusable. Specific examples are: assumptions about the execution environment, hardware performance, memory, garbage collection, and architecture assumptions, and "tricks" that make it less portable in some way (e.g., machine dependencies). In theory, this author review could be extended to a full set of subjective opinions by the author as well, however this may be impractical and unnecessary because of the author's inherent bias.

8. **Generic Parameters** – Classes which are generic in function are often parameterized on some other type (or types) which further specifies the behavior of the class. Such classes describe a common implementation of a data structure applied to various types of objects, where this type is supplied as a parameter to the class. Note that not all object-oriented programming languages support this capability.

   Parameterization increases the reusability of a class because it defines a higher level interface to data elements which can be specified by the reuser of the class. The generic parameters of a class are therefore stored in the ORT library so that they can be properly documented and matched against. Parameters are documented by a name and a textual description. Because generic parameters are inherently polymorphic (i.e. the parameter can be bound to any class which satisfies the interface required by the parameterized class), there is little information that can be matched to a specific generic parameter. Their purpose is primarily for documentation of the class, although queries in ORT do provide the ability to specify parameters.

The entity-attribute-relationship model in Figure 4-1 succinctly conveys the information stored by ORT. However, it was deemed unreasonable to attempt to use an object-oriented or entity-relationship database in the realization of this model. The problem with either of these approaches is not their applicability, but rather their lack of practicality. Many such databases are special-purpose, unproven, and generally not widely used as yet [15, 9].

Instead, the model has been translated into a relational model so that it may be implemented using the ANSI/ISO-standardized Structured Query Language [12, 1], which is better understood, more robust, extremely portable, and affords transparent performance increases and distribution between hosts merely via upgrading to one of the enhanced SQL products. Details concerning the SQL implementation are left to Chapter 7.

4.3 An Example Class Description

Before providing an example of a Class Description, yet another representation of its contents must be introduced. The *flat-file representation* of a Class Description contains a single class entity and all the information linked to it. It is a stream of characters, suitable for storing in a file, in a simple report format. The purpose of the flat-file representation is to provide a format in which Class Descriptions can be extracted from the library, edited by the user for re-submittal, entered into the library, and moved between libraries. The definition of the flat-file format is best described by example.

The following example is the flat-file version of the Class Description for a tree abstraction written in a version of C++ with parameterization[1]. Note that the syntax of the form is language-independent, save for the information about portability. Also note that the information is partitioned into logical *fields*. As will be further explained later, these are not individual data fields in the database, but groups of related information that are

[1]This class is based roughly on the tree class for Bertrand Meyer's Eiffel [33].

considered basic units of storage by the classes that interface ORT to the database. This grouping of data is represented in flat-file format with a field header and trailer marking the beginning and ending of fields.

```
################################################################################
#                          Main Class Information:                             #
################################################################################

Class Name:  Tree

Revision:  1.0
Revision Time (YYMMDD.HHMM):  891024.1022

Short Description(<100chars):  Generic Tree abstraction.

Contact:  morgan@wh.ai.mit.edu

Type Of Entry:  S

Total NCSS:  209
Lines Of Documentation (for class):  54
McCabe Complexity (total):  15
Number of Class Functions:  8

OS ID:  10
Machine ID:  8
Compiler ID:  2
Project ID:  46

Number of Class Parameters:  1

################################################################################
################################################################################
#                            Class Description:                                #
################################################################################

Class Tree implements crude tree functionality and is intended to
be used as a base-class for more sophisticated and specialized
tree classes.

No distinction is made between trees and tree nodes.  Each node
is a tree containing one or more elements of type T, the leaf type.

Each tree includes a ''cursor'' which may be moved by various
member functions.  The action performed by some member functions
depends on the current position of the cursor relative.  The node
where the cursor is currently positioned is called the current node.

################################################################################
################################################################################
#                            Class Parameters:                                 #
################################################################################

Parameter : T

The leaf-type of the tree.

################################################################################
```

```
##############################################################################
#                              Associated Files:                             #
##############################################################################

FILE : Specification/Include File
/bfd/ort/files/generic/Tree/Tree.h

  This is the standard C++ .h file.  It can be multiply-included as
  it is formatted:

  #ifndef TREE_H
  #define TREE_H
  <specification of class>
  #endif TREE_H

  to insure that a compilation never sees the specification twice.

FILE : Source Code File
/bfd/ort/files/generic/Tree/Tree.c

  This file contains the entire implementation of the class.

FILE : Test Code
/bfd/ort/files/generic/Tree/testTree.c

  This file contains a simple test suite for class Tree.

##############################################################################
##############################################################################
#                               Class Features:                              #
##############################################################################

Feature : Constructor
Tree<:T:> : Tree<:T:>
 ( )

  Constructs a null tree (an empty root node only) with the cursor
  on the root.

Feature : Constructor
Tree<:T:> : Tree<:T:>
 ( Tree<:T:> : t )

  Constructs a new tree which is a shallow copy of t.

Feature : Observer
Tree<:T:> : value
 ( )

  Returns value of the current node (where cursor is currently
  positioned).  If current node is a leaf, return null.
```

```
Feature : Observer
T : leafValue
 ( )
```

  Returns value of current node. If node is a leaf, return null.

```
Feature : Observing Mutator
Tree<:T:> : changeValue
 ( Tree<:T:> : t )
```

  Changes current node to t, returning
  the node's former value (null if it was a leaf).

```
Feature : Mutator
void : addLeft
 ( Tree<:T:> : t )
```

  Inserts t between the current node and its sibling to its left.
  If no such sibling exists, then it is inserted before (to the left)
  the current node.

```
Feature : Mutator
void : addRight
 ( Tree<:T:> : t )
```

  Inserts t between the current node and its sibling to its right.
  If no such sibling exists, then it is inserted after (to the right)
  the current node.

```
Feature : Mutator
void : delete
 ( )
```

  Deletes the current node and places the cursor on its parent.
  If the current node is the root, then the result is the same
  as Tree<:T:>().

```
Feature : Observer
int : arity
 ( )
```

  Returns the number of children that the current node possesses.

```
Feature : Observing Mutator
Boolean : moveToChild
 ( int : n )
```

  Moves the cursor to the nth child (from the left) of the
  current node. Returns true on success, otherwise false.

```
Feature : Observing Mutator
Boolean : moveToParent
 ( )
```

  Moves the cursor to the parent of the current node. Returns
  true on success, otherwise false.

```
##############################################################################
##############################################################################
#                   Class Implementation Depends Upon:                       #
##############################################################################

LinkedList   2.3

##############################################################################
##############################################################################
#                          Class (Re)Use History:                            #
##############################################################################

Successful :  891025.0933
   Contact :  morgan@wh.ai.mit.edu
    Rating :  8
     %Mod. :  0

  Original use of class by author to demonstrate.

##############################################################################
##############################################################################
#                                  Keywords:                                 #
##############################################################################

Description : tree
Description : leaves
Description : leaf
Description : node
Description : cursor

Feature : tree
Feature : value
Feature : leafValue
Feature : changeValue
Feature : addLeft
Feature : addRight
Feature : delete
Feature : root
Feature : arity
Feature : moveToChild
Feature : child
Feature : moveToParent
Feature : parent
Feature : node
Feature : nth

##############################################################################
```

```
################################################################################
#                                  Reviews:                                    #
################################################################################

Review : 890313.1411
Reviewer : cohn@hpwarl.wal.hp.com

Specificity : 2
Overall Quality : 8
Functionality Documentation Quality : 5
Documentation On How-To-Use : 3
Quality Of Interface To Other Abstraction Levels : 8
Support For Debugging : 6
Separation Between Code and Specification : 7
General Readability of Code : 9
Generality of Class : 8
Clarity of a Semantic Role : 9

################################################################################
################################################################################
#                                 Comments:                                    #
################################################################################

Comment : 890313.1411
Commenter : cohn@hpwarl.wal.hp.com
Type : Review Comment

This is a reasonably well thought-out minimal tree abstraction.  It is
simple and easy to learn,  but lacks in general usefulness because of its
simplicity.  What there is of it is well done, but a more extensive
interface may be appropriate so as not tax the programmer as much.
Needed functionality consists of the following features:

 * A way to move the cursor right and left
 * Observing functions like isRoot, isLeaf, etc.
 * A way to mark a node and return to it after the cursor has been
    moved

################################################################################
```

This chapter contains a description of the functionality which has been prototyped to demonstrate the effectiveness of ORT. This consists principally of ORT itself, which provides the major benefit of the effort associated with entering the mass of information described in the previous chapter. There is, however, some additional functionality, in the form of two other tools, CHECKIN and CHECKOUT, which have been implemented to facilitate the loading and editing of data in the library.

5.1 Querying with ORT

There are two basic approaches that can be used in querying. Either a query can be an entire Class Description for a class desired by the user or it can be an expression of only those class traits which the user cares to specify.

If the representation of the query uses a simple query language, then queries need only contain the specification of properties desired by the user. The interface is greatly simplified, because long forms of mostly blank entries can be replaced with simple boolean expressions. Additionally, a query language can specify more clearly what is desired by using set notation and explicit relational operators. The ideal solution may be to offer both of these approaches to querying, but the query language approach was selected since it is a more intuitive and takes advantage of ORT's browsing capabilities by encouraging vague queries.

ORT allows the user to specify a collection of library entries in a query language Q. Q is a simple, specialized input language designed to allow characteristics of classes to be specified as a standard boolean expression using the AND, OR, and NOT operators, as well as explicit grouping with parentheses. Terms in this boolean expression are of the form:

$$match\_field \ \ relation \ \ value$$

where *match_field*, *relation*, and *value* are as specified below.

The *match_field* of an expression in Q specifies a single data element or a set of data elements from the Class Description of classes which match for this term in the expression. The possible *match_field*'s are a subset of the attribute fields in Figure 4-1. Though more sophistication could be added, including, for example, the matching of feature templates, these pieces of information are selected as the most effective in specifying collections of relevant classes.

In the cases where *match_field* denotes a set of keywords, it corresponds to a collection of classifying terms which have been chosen for a particular facet of the class. Currently, two types of keywords can be specified with Q. These two keyword types correspond to the search-fields "Short Description" and "Feature Description." Keywords

are segregated to focus the search process and give the keywords contextual meaning. Other keyword types can be added by augmenting the language definition for Q. These two types were chosen because, in the first case, they indicate the role of the class or the concept that the class represents and, in the second case, they indicate what can be done to a class, what properties a class possesses, or what functionality the class delivers.

A *match_field* can be one of the following:

- Portability constraints (i.e. *machine, os, compiler, project*) which correspond to attributes of the class entity.
- *desc_keys*: The set of keywords selected from the description of the class. These keywords are one type of keyword as defined by the entity in Figure 4-1 (i.e. the search field is the description attribute for the class entity).
- *feature_keys*: The set of keywords selected from the description of a feature. These keywords are used and treated similarly to, but separately from, the previous class of keywords because the associated search field is the description.
- *name*: The class name attribute of the class entity or an alias for this name.
- *contact*: The contact attribute of the class entity.
- *params*: The set of generic parameters for the class (name attribute only).
- Class metric measurements (e.g., *size, complexity*) which are attributes of the class entity.
- *date*: The date portion of the revision time attribute of the class entity.
- *entrytype*: The type of entry attribute of the class entity (either "submitted" or "unsubmitted", but almost always just "submitted").

The *relation* is one of: $=, <, >, <=, >=, elt\_of$, or $intersects$, each of which has the standard semantics. Note that the use of elt_of is a shorthand for a set of clauses combined with the OR operator.

The *value* in a Q expression can be a number (if the associated $match\_field$ contains a metric), a date of the form MM/DD/YY (for the *date* $match\_field$), or a set expression, which is a set of alphanumeric character strings separated by commas (e.g., $\{mouse, cursor, sprite\}$).

An example of language Q is given in Figure 5-1. It contains the default query that the user can edit by adding more specifics and deleting null clauses.

It should be noted that ORT has no true classification system. Rather than subscribing to an arbitrary enumeration over the entire set of domains in which software is applied and then forcing classes to be somehow pigeon-holed into a slot of this enumeration, ORT takes a less structured, but more flexible approach. It is anticipated that most relevant entries will be found through the use of keywords.

/tmp/OrtQuery

```
#
# Default Query for ORT
#


QUERY:

     entrytype elt_of {submitted}
and (
     name elt_of {}
  or params intersects {}
  or (
          date >= 01/01/87
      and contact elt_of {}
      and complexity <= 15
      and size <= 1000
      and (
             desc_keys intersects {}
         or feature_keys intersects {}
           )
      and machine elt_of {}
      and os elt_of {sysV, bsd, SunOS, HP-UX, Ultrix, Mach}
      and compiler elt_of {C++2.0, C++1.2}
      and project elt_of {}
      )
    )
```

-----18.44.2 OrtQuery   (Text Fill)--Top----------

Figure 5-1: An example query.

When the user invokes ORT, he or she is placed in an editing window in which a default query, or query template, is pre-loaded. The desired query can then be developed and saved in a temporary file pre-associated with the query. The user can then choose to execute the query, re-edit the query, or quit ORT. If the query is valid, its execution will place the user in the browsing phase, described in the next section. Otherwise, an appropriate error message is issued and the user is placed back in the editor to correct the Q expression.

The default query mentioned above is customizable. An example of such a query is shown in Figure 5-1. Normally this query comes from a system-default file, which can be customized to the needs of the users on the system. For example, the definition of "too large" or "too complex" is essentially held by these defaults and will rarely get changed. Additionally, portability constraints may be reasonably constant for those doing development at a single site. The default query can be further customized on a per-user basis by copying the system-default file into a user's default file (called ".ort" in the user's home directory) and editing it to suit personal tastes.

5.2 Browsing with ORT

Browsing is based on the concept of ORT Browsers, which were introduced in Chapter 3. The result of executing a valid Q expression is the primary class browser, from which all other ORT Browsers are generated. These ORT Browsers are windows in the X Window System [17], which has become a common part of current software development environments. These windows (sometimes referred to as *widgets* in X-Window terminology) contain four visual components: a title and header, a menu of entries, a scroll-bar, and a set of eight buttons. An example of a class browser is shown in Figure 5-2 as it might appear on most any vendor's X-Window display terminal.

The browser title and header describe what the browser is and what the contents of the menu are. The menu contains textual, one-line entries that display part or all of the collection of entries associated with the browser. The scroll-bar is used to page through the browser entries when they can not all be displayed at once. At all times, there is a current selection, highlighted with color, which can be changed via paging or by clicking the mouse on the desired entry. The buttons across the bottom of the window are function keys F1 through F8. They can be activated via keyboard or by clicking the mouse on them. These buttons model *soft keys* (commonly found on HP products). Soft keys are keys which are always present, but whose function can be changed to suit the context of the application.

There are a set of conventions associated with the functions available through ORT Browser soft keys. The left-most key always provides context-sensitive help, describing the functions of the keys and the purpose of the browser. The right-most key always terminates the browser and, in the case of the primary browser, terminates the browsing phase of ORT. The functions associated with the remaining soft keys usually generate either a text viewing window containing the requested information or another Browser

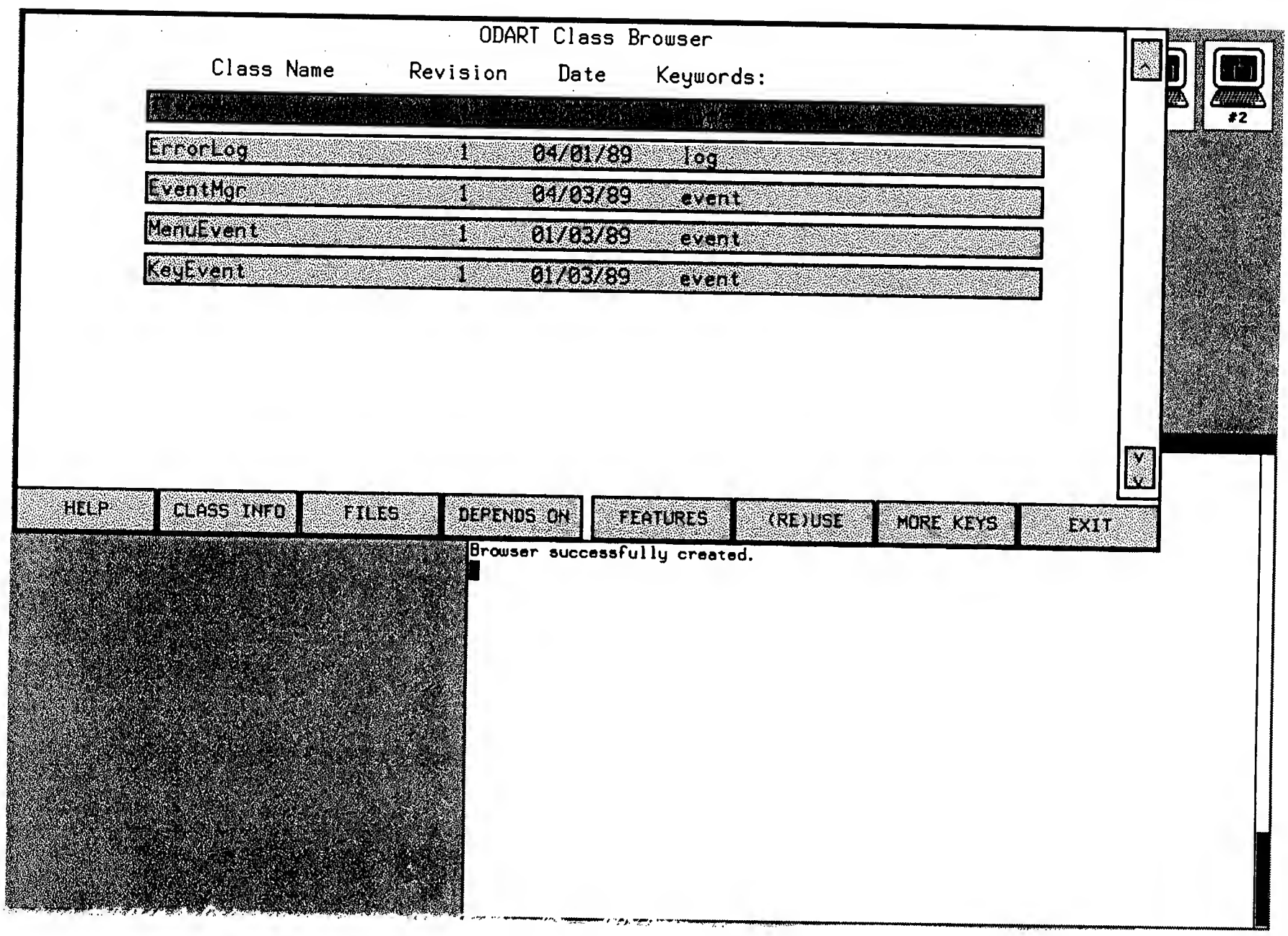

Figure 5-2: A Class Browser running with other X-Window applications.

Widget. In cases where more than six such functions are required, a "MORE KEYS" key is provided next to the exit key which will map in other keys available to the user. ORT Browsers come in four flavors:

1. **Class Browser** – Nodes in the search for relevant classes in the ORT library can be represented by *collections* of classes, which are presented in class browsers. Users may proceed from one node in the search tree to others by selecting a class and an operation on that class (for example, "DEPENDS ON", to generate all classes needed to implement the selected class) to produce another class collection.

2. **Feature Browser** – Similarly to the Smalltalk browser [22], class features can be browsed in ORT (although they can only be viewed, not modified). Feature Browsers allow the user to browse this collection of features, which can be associated with a class either by direct definition or inheritance.

3. **History Browser** – The history of successful and unsuccessful usage attempts associated with a class can be examined through the use of a history browser.

4. **File Browser** – The files associated with the implementation and verification of a class are directly accessible from within ORT. Interactions with the file system are managed via a file browser, which allows files associated with a class to be viewed, edited, or copied to a local location in the file system for subsequent reuse or leverage.

One final function, accessible only from the primary class browser, is the ability to revert back to the query that began the browse. Should the user deem a browse fruitless, the "RE-QUERY" key terminates the primary browser and allows the user to re-edit the most recent query in hopes that there is some change that can be made to the query to allow more relevant classes to be selected.

5.3 Library Maintenance Tools

Because of the large-scale transactions that must occur when entering a Class Description into the library or editing its contents once there, two tools were developed to facilitate such transactions during the prototyping and demonstration of ORT. These tools, CHECKIN and CHECKOUT, allow form-based transactions to occur between the library and the flat-file format (described in Chapter 4).

CHECKOUT invokes an *unparsing* process in which a Class Description is converted from data stored in the library to flat-file format. The user can then add, modify, or delete information under the constraints imposed by the format. Because the editing process may take a number of hours or even days, a lock can be placed on the Class Description (assuming it is not already locked) to prohibit simultaneous edits. At any time, a Class Description may be checked out for examination purposes (i.e. with no lock).

CHECKIN invokes the complimentary *parsing* process on a flat-file, performing appropriate error-checking before committing data to the database. Should the check-in process be successful in parsing and in obtaining permission to store data to the library, then a new Class Description is entered into the library as a newer version of an existing one. In this way, version control is implemented on the documentation stored in the library (for the same reasons as is currently done with source code files).

CHECKIN can fail to obtain permission to store a Class Description in the library under two circumstances. If the user executing CHECKIN is not on an access-control list for the ORT library, then modification of any part of the library is prohibited (the user may not lock a Class Description either). The access-control list is intended to include only those who are directly responsible for the maintenance of the library. This can be a small, select editorial committee or a list of all software engineers developing object-oriented software locally. The other situation in which CHECKIN will fail is that in which a lock has been placed on the particular Class Description being stored by another user. In such a case, the other user must first check-in his or her changes before anyone else can check-in their changes. Presently, merging of different versions must be done manually by editing flat-files.

6. Demonstration of ORT

This chapter uses a scenario to illustrate how ORT can be used in an object-oriented software development environment. It is important to note that the designer, not the programmer, is the user of ORT. It is the designer who should not only be sensitive toward maximizing reuse but also make the compromises in the design toward that end. It is not appropriate for the implementor to re-think a design so that he or she can reuse more code than the current design dictates, nor is it appropriate for the designer to construct a design in a vacuum, free of the information presented and maintained by ORT.

The data-accessing capability of ORT is the payoff of much data-entry. The ability to access information in a library requires much advance work. This effort results in a set of important benefits not provided in traditional development methodologies. This scenario demonstrates these benefits, listed below.

1. **Assistance in Assessing Reusability.**

   A query submitted to ORT may result in the suggestion of many classes for potential reuse. These classes will vary in their semantic proximity to what is actually desired, their semantic adaptability, their design quality and adaptability, and their implementation quality and adaptability. It is important to be able to quickly determine *which* to reuse as well as *whether* to reuse. The following types of information stored in the library help in assessing the reusability for this collection of classes:

   (a) judgments of portability

   (b) measurements of code metrics

   (c) history of reuse attempts

   (d) commentary on reusability

   (e) reviews by an editorial committee

2. **Querying Stimulates Design Improvements.**

   Because of the keyword-based classification scheme, queries can lead to results which stimulate the user to discover possible design improvements. This can occur in two ways:

   (a) A keyword representing a general concept can generate a collection of entries that may expand the ideas present in the design. This introduces new ideas into the design process early and encourages the designer to more carefully choose terms and names.

(b) Words that appear in the query can trigger the use of the correct generic classes. Though any small collection of generic classes can be managed without tools, quickly arriving at the correct ones is important. Using ORT, it is more likely that generic classes will be selected based on the conceptual description of the desired class than on the whim of the implementor. Examples of such terms are: queue, key, mapping, mapped, and linked.

3. **Non-local Browsing.**

   ORT offers many links to follow between its library entries. This means that there are usually many ways in which to proceed at any given step in a browse. A complicated browse can take the user conceptually far from where the original query had intended. This expands the flexibility of the search for reusable classes with ORT.

4. **Identifying Possible Design Modifications.**

   To avoid back-tracking and designing-out reuse, the search for potentially reusable components should occur while design changes necessary to incorporate relevant library entries can still be made easily. ORT's use can implicitly suggest such modifications. In fact, the overall goal of ORT is to suggest design changes in situations where the design could be made slightly less elegant but more standardized, saving substantial implementation time.

   Aside from this overall benefit, there is a special case where significant added benefit is realized. If ORT is being applied in a domain that is well understood by those populating the library, then it is likely that the library holds a fair amount of knowledge about programming in this domain. In such a case, the lack of results of a query and browse session suggests to the user that the desired class may be a poor choice of abstraction in this domain. Assuming that the editorial process used to approve ORT library entries filters out poor abstractions, designers will not find poor choices in the library. The lack of results should suggest to the user to re-think the design and respond by either asserting that the new abstraction is truly useful and worthy of original implementation work or by modifying the design to use more standard components.

6.1 Initial Design Ideas

Designers evolve their notions of design in an incremental fashion, by first expressing a broad framework, which is the most reusable part, and then filling in the details, possibly reorganizing the broad framework somewhat in the process [5]. As a result, careful attention should be paid to the level of detail provided in producing queries from tentative design descriptions.

It is assumed that before the start of the scenario enough analysis and initial design effort has been expended to produce the following informal and, as yet, "half-baked"

description of a system's top-level abstractions. The results, whether produced using a state-of-the-art CASE design tool or merely "back-of-the-envelope" techniques, are intended to represent a believable first pass at characterizing and decomposing the intended system. The designer's ideas, as they might appear in a design notebook or repository, follow:

> A new simulation system is desired to model the activities of an organization containing chains of command. Input events are introduced into the organization, each of which can trigger any number of internal activities, as well as any number of outputs. Rather than help describe the processes in an organization, this system simulates the dynamic behavior of the organization while it undergoes changes in structure. The motivation for building such a program is to provide a tool to help train managers of organizations that can be modeled effectively by this approach.
>
> **Organization**: An organization is specified by: sources of input, types of output, the components of the organization, and the chains of command connecting these components. The chains of command are modeled using a graph, where nodes represent the organization's components and chains are expressed as arcs between them. Activities within the organization are modeled by events that pass between the nodes along arcs. The organization abstraction represents the entire system; an execution of the program consists merely of an organization creation, simulation, and destruction.
>
> **Node**: A node in an organization can be a person, a processing service (like a computer system), or a repository (like a filing cabinet). Nodes have labels, describing what they are, and a function called processEvent, which is written by the user and describes what the node does. Nodes also have event input queues; only one event may be processed at a time, taking an indeterminate period to complete. Upon dequeuing of an event, a node can discard it, forward it to other nodes to which it is connected, or compute a new set of events to originate.
>
> **Arc**: An arc is a relationship between two nodes, one of which directly precedes the other in some chain of command. Arcs transport events from an originating node to a receiving node.
>
> **Input**: An input source for an organization introduces events into an organization. It is connected to nodes via arcs but can not be a receiving node on any arc. Further, its processEvent feature is called every clock "tick" by the simulation system, optionally emitting an event based on a user-defined algorithm.
>
> **Output**: An output destination for an organization collects events from an organization. It is connected to nodes via arcs but may not be an originating node on any arc. Because they are outputs of the organization, events are collected for examination by the user,

Event: An event represents a transaction between two nodes. This activity is governed by the processEvent function defined for the node. Events have timers associated with them, so that the time it takes for them to be processed can be computed. Events also have logs in which nodes can store information destined for other nodes.

Menu: A menu-based user interface allows the user to control the organization. A menu structure containing the following capabilities is currently anticipated:

- node: create, label, set processEvent function, delete, examine queue, clear queue, activate, deactivate
- arc: create, label, set event originator, set event receiver
- quit: exit, print statistics and exit

6.2 The Contents of the ORT Library

For the purposes of this demonstration, there are only 31 entries currently in the ORT library. Only a fraction of these entries are relevant to the organization simulation system. The others must be filtered out, at least partially by ORT. The following is a representative sample of the classes currently in the library.

- Queue – A generic class representing the canonical queue data structure.
- EventMgr – Part of a system used to manage resource allocation, this class centralizes and dispatches events from the system to the correct agents.
- Graph – A class representing a simple graph, with nodes and arcs between them. It has been found useful in building higher level classes based on graph theory.
- DirectedGraph – An heir (sub-class) of Graph, this class replaces the notion of undirected arcs with that of directed arcs.
- DAG – An heir of DirectedGraph, with the added restriction that the directed arcs can not form cycles.
- DagWidget – A class based on DAG, but also a user-interface object, in that it displays itself (i.e. a directed, acyclic graph) in its own X-Window.
- Timer – A simple timer that can be started, stopped, reset, and read.
- ErrorLog – An error log useful in products where system errors should be logged for later examination.
- Browser – A browser containing a menu, a menu scroll-bar to page items through the menu, and a set of soft-keys. It manages user interactions and provides a call-back mechanism to invoke user-supplied functions upon activation of soft-keys.

- KeySet – A set of 8 keys that can be loaded into a Browser's soft keys. It keeps track of KeySet level help strings, individual key help strings, key labels, and key codes.

As it happens, there is little chance for reuse of earlier work at the top level of abstraction (the organization). This is not surprising, since this particular approach toward simulation is not likely to have been implemented and entered into the ORT library. The next few levels of abstraction (i.e. the rest of the vague ideas in the design description), however, still offer promise for attempts to reuse previous work.

6.3 Querying and Browsing with ORT

In the scenario, the designer begins with the characterization of the intended system based on the vague ideas shown above. The immediate goal is to quickly convert these concepts into concrete classes that not only represent the "right" set of abstractions, but can be implemented quickly thanks to reuse. It is unlikely, however, that each of the informal abstractions will directly map to a class in the system being implemented. Each idea is nevertheless converted into an ORT query to find classes similar to what the idea represents.

Queries are submitted to ORT to help refine the set of desired classes into a design that reuses entries in the library. This is done by using a query to generate an initial set of classes from the library, then browsing until either an appropriate choice for reuse is made, or it becomes evident that no such choice can be made.

The user begins by constructing a query for the first potential class in the design (see Figure 6-1). As should be customary with ORT, care has been taken not to over-constrain the search by changing too many terms in the standard *Q* expression. The standard system-default and user-default .ort files contain very general, or *lenient*, *Q* expressions. The query for the "Node" abstraction is a modification of the user's standard query, which looks for classes written in C++ and run under some flavor of the Unix operating system (this query template was shown in Figure 5-1). The changes involve adding keywords to specify classes related to the user's current understanding of "Node".

Queries should be kept lenient, but not *too* lenient. Each query must contain some restricting information or a search of the entire library would ensue. The designer should describe, especially at the higher levels, the desired abstractions needed to model his or her problem. The process of designing with maximal reuse should not break down into the process of finding a problem for which a solution is known. Rather, original implementation must be done at some level, but that implementation should be expedited by maximizing intermediate- and fine-grain reuse.

The class browser shown in Figure 6-2 is generated from the query in Figure 6-1 and contains only a few matching classes. As might be expected, no truly applicable classes match the concept of an organizational node. However, the user benefits in two important ways.

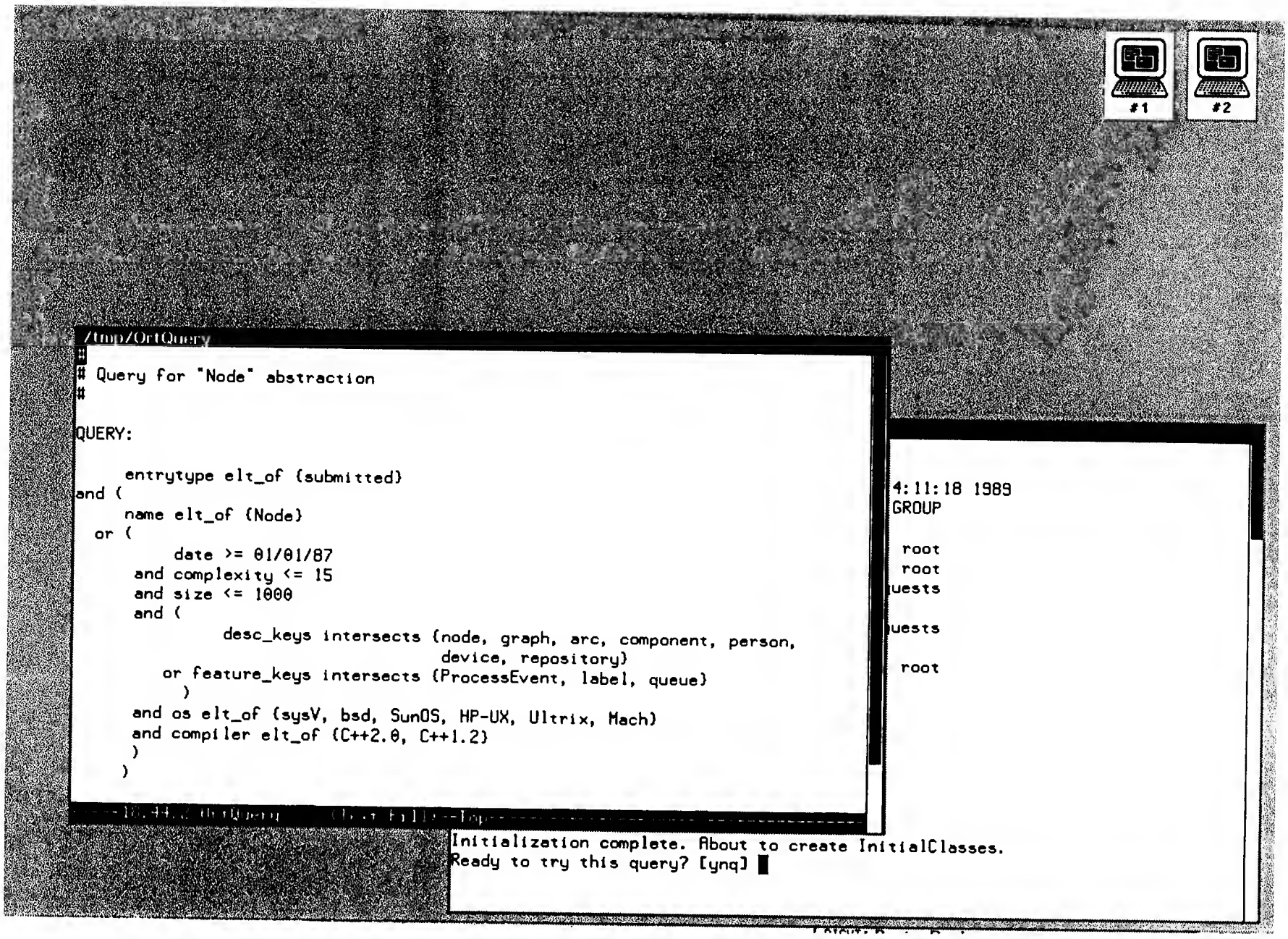

Figure 6-1: User edits query for "Node" abstraction.

First, some relevant ideas about what nodes can model in an organization (from a computer-automation standpoint) are derived from this query. The user can make a mental note that in the final system, a library of nodes should exist, some of which represent things like printers, disk-drives, and other mass-storage devices. This is a specific example of one of the general benefits of ORT described at the beginning of this chapter. By querying a library of software components, a general term in the keyword collection (i.e. "device") generated classes which, by mere mention, have expanded the current design. In other words, wrong choices may still be useful for other parts of the design.

Second, a class has been suggested that can be useful in the implementation of class Node, should it be necessary. Not surprisingly, a class Queue, written in C++, exists in the ORT library. It was matched by ORT as a result of the inclusion of the keyword "queue" in the "feature_keys" clause of the query. This is an example of the second major

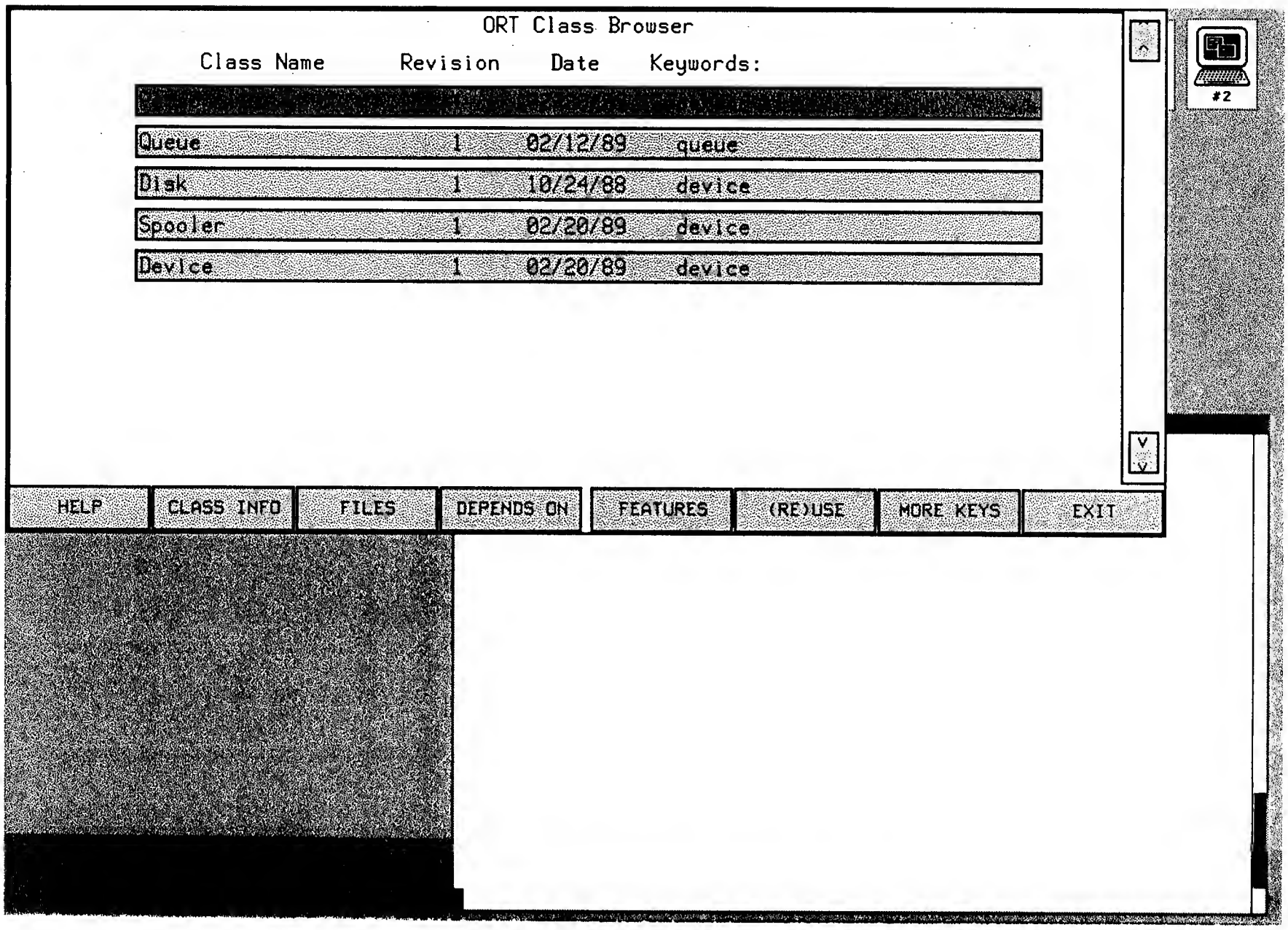

Figure 6-2: Class browser containing relevant candidates for "Node".

benefit of ORT, where a relevant generic class has been selected by converting the textual description of "Node" into a set of keywords.

Before proceeding, the novice user utilizes the context-sensitive help facility. A help window, shown in Figure 6-3, is generated to explain what the visible soft keys of the primary class browser do. This was done by clicking the mouse on the the HELP soft key.

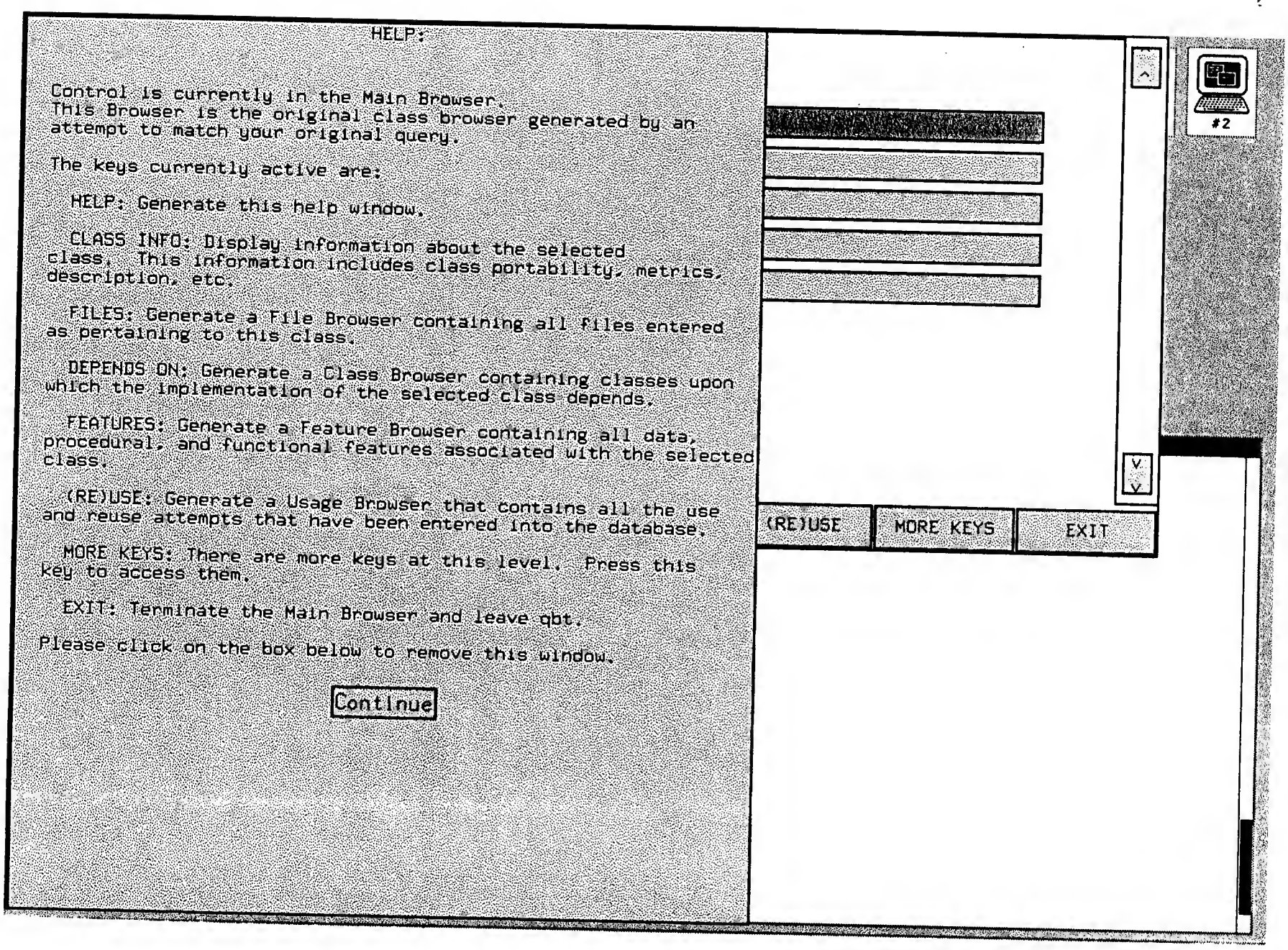

Figure 6-3: Help window for initial keys on the primary class browser.

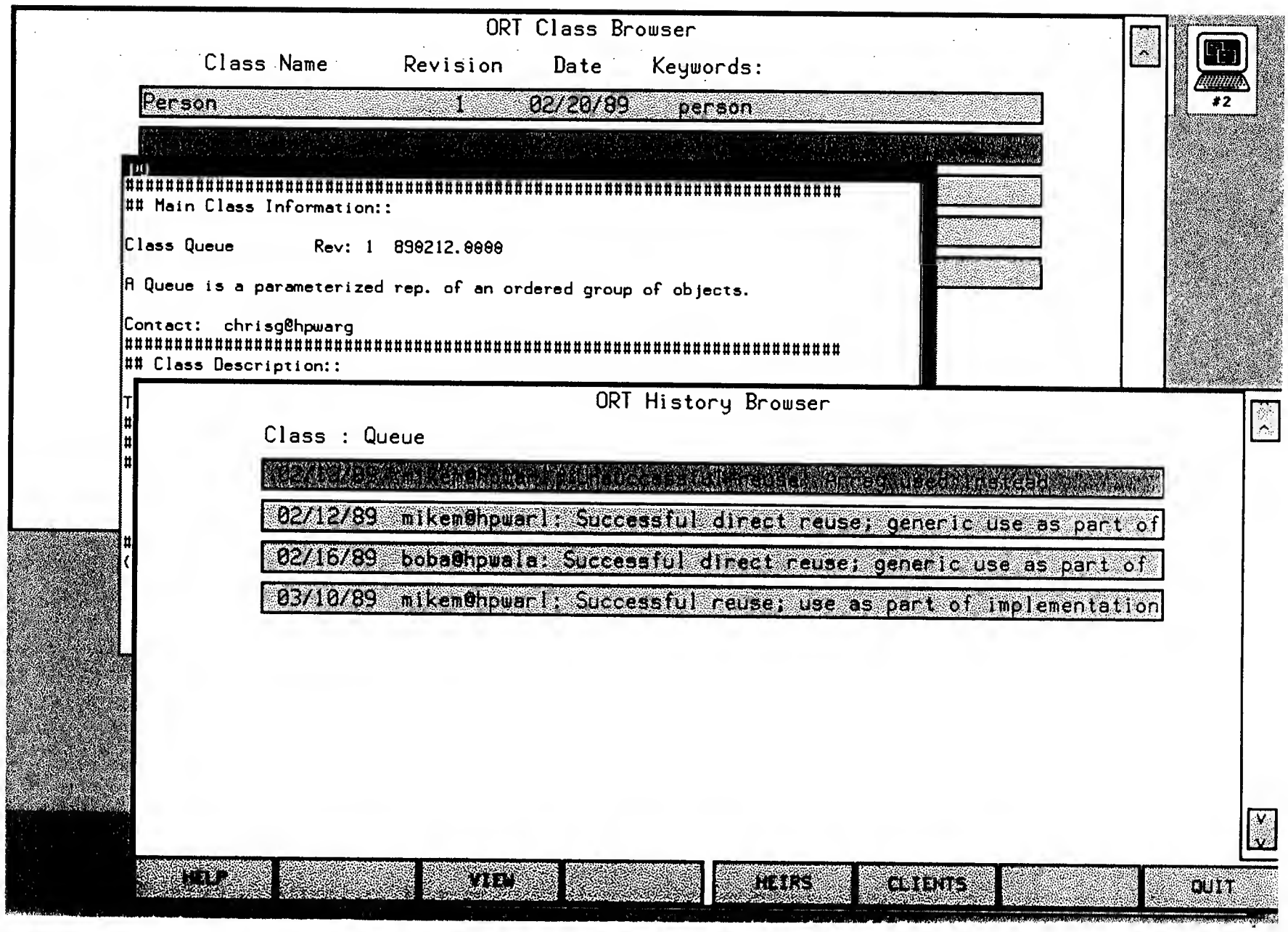

Figure 6-4: Class Information and History Browser for class Queue.

Since the Queue class appears important, ORT is used to verify that Queue is reusable in the context of the intended application. The user chooses the Queue entry, then generates a class information viewing window (with the CLASS INFO key), producing the second window, layered upon the first. Finally, the user generates a history browser with the (RE)USE key, producing the top-most window. The result is shown in Figure 6-4.

The user finds that class Queue has been reused by a number of others. In the class information window, which contains most of the information in the Class Description not available via a browser, the user finds that care was in fact taken to implement and document this class because it represents a generic, and therefore highly reusable, data structure. As described in the first major benefit of ORT, the user has been assisted by the library's information concerning reusability.

Because no closely matching library entries were found on the "Node" browse, the

user now realizes that Node must be implemented as a class. It also becomes apparent that there is a common element between "Node" and "Output": a queue of events. Further, both "Node" and "Input" have user-definable procedure features which control how events are manipulated by an organization. At this point, the user decides that the various types of nodes should make up an inheritance hierarchy in which the most generic "Node" functionality can be implemented in a base class Node and specific types of classes can inherit from it, specializing parts of the interface as needed. The direct heirs of Node would be InputNode, OutputNode, and ComponentNode. Nodes actually used in a simulation would then, in this new scheme, be heirs of one of these three classes.

Despite these design changes stimulated by ORT, the user continues in the search for more reusable classes before moving on to more advanced design stages, since additional design changes may follow. The browsing phase of ORT is exited by removing all windows (in this case, clicking on QUIT in the history browser, hitting 'q' in the pager displaying the class information, a finally hitting EXIT in the class browser). ORT is re-run with a new query, this time in search of relevant classes for the "Arc" abstraction. The query, shown in Figure 6-5, is constructed, using the keyword fields to specify the concept.

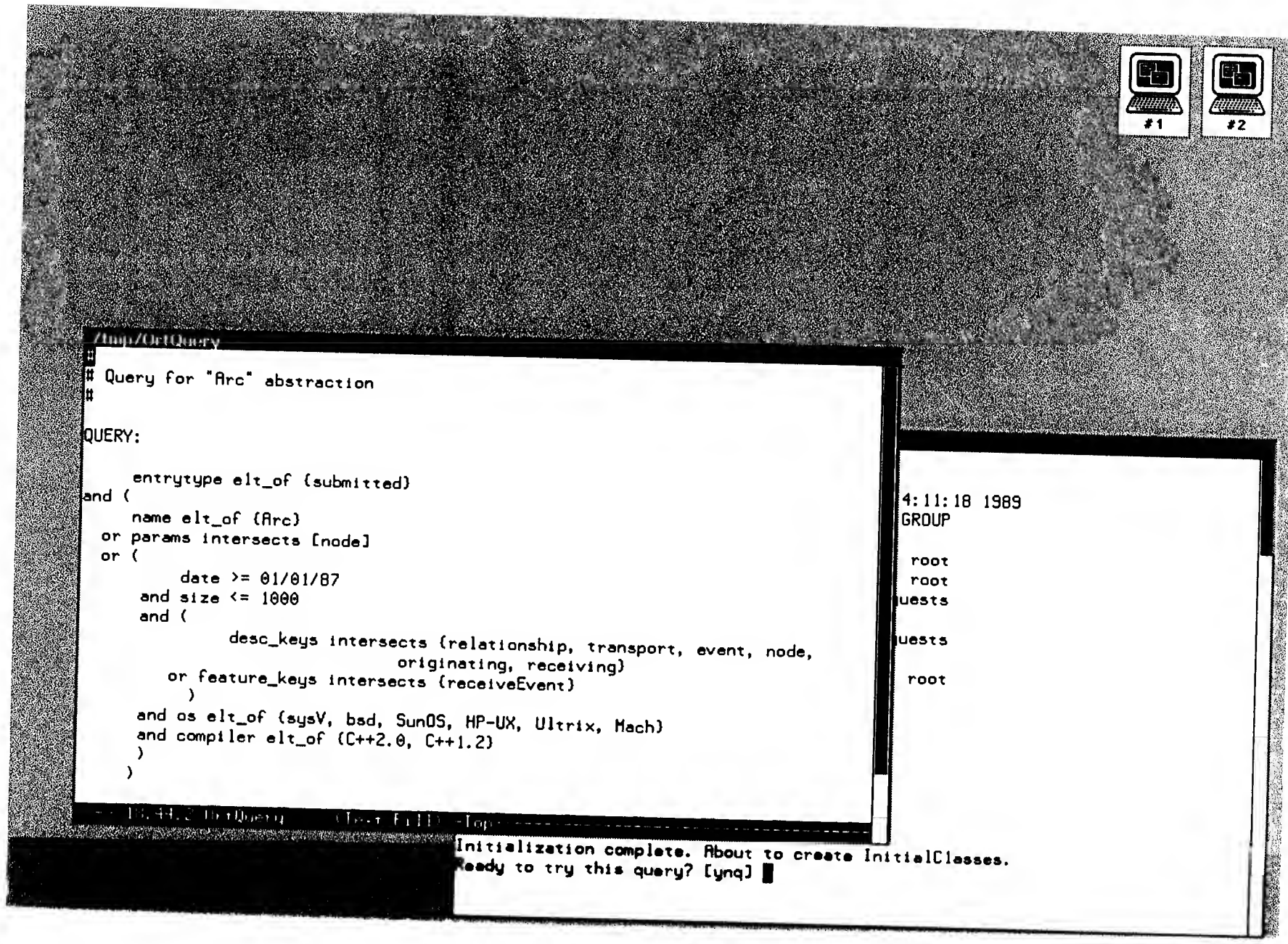

Figure 6-5: User edits query for "Arc" abstraction.

The resulting browser (which appears as the upper window of Figure 6-6) only contains two classes: Link and Graph. The Link class, upon investigation (not shown), is found to be part of Keith Gorlen's C++ Classes [23]. It is a class "used to construct LinkedLists" and is thus not applicable. The other entry in the class browser, however, is of significance: Graph could be used to represent an "Organization".

The user realizes that a DirectedGraph class would be more appropriate, since Events move in only one direction between Nodes. To investigate this possibility, there are two alternative approaches: (1) the user can use keys "MORE KEYS" and "RE-QUERY" (currently not available) to adjust the original query or (2) the user can attempt to find a DirectedGraph class via browsing. Since DirectedGraph is likely to be in the same inheritance or dependency hierarchy as Graph, the user proceeds to investigate the parents and heirs of class Graph.

Before continuing, however, it is useful to digress long enough to illustrate why the

original query failed. It did not match DirectedGraph, which exists, because there was no "graph" keyword and the description of DirectedGraph (from which these keywords should be taken) is "...like a Graph, except that arcs have a direction associated with them." Thus, by adding the more general "graph" keyword to the "Arc" query shown in Figure 6-5, more candidates for reuse could have been obtained.

Note that, essentially, the use of ORT has stimulated changes in the goals for reuse. The designer now has a better idea of what an Organization is: a DirectedGraph of Nodes. This is a specific example of the final ORT benefit described initially, *identifying possible design modifications*, in that a lack of results implicitly caused the designer to re-think the current design conceptualization.

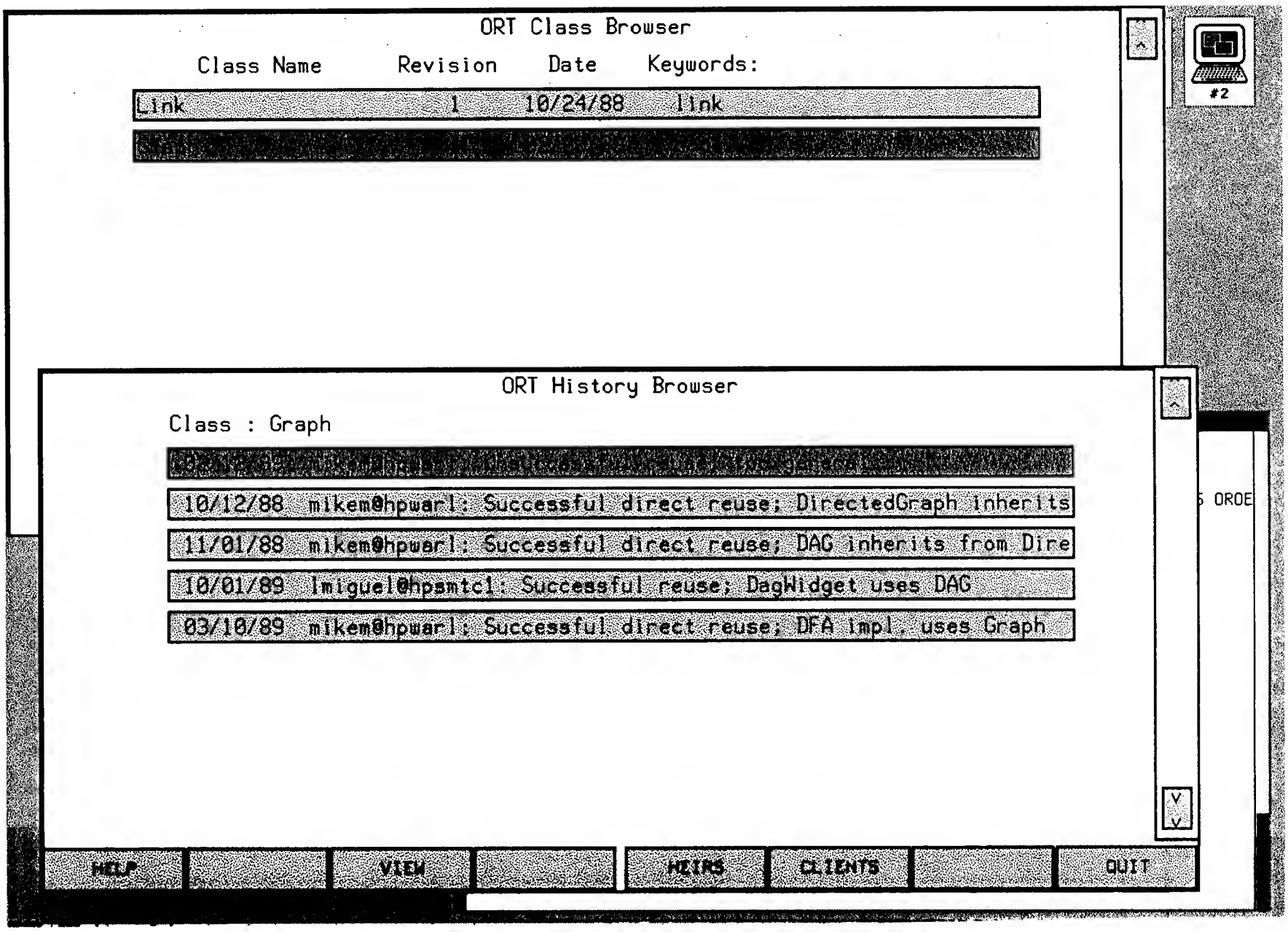

Figure 6-6: The search for "DirectedGraph" (Part 1).

The user investigates "up" the hierarchies through use of the History Browser. All uses of a class, via direct use or inheritance, are contained within the context of a reuse of the class. However, what one project chooses to do with a class may not be what another project does with it. Thus, heirs (sub-types of this class) and clients (those depending on this class) can only be accessed through the history browser (see Figure 6-6), which is generated by selecting the entry for Graph in the class browser and clicking on the (RE)USE key. A successful usage attempt can then be selected and the HEIRS or CLIENTS key used to generate additional class browsers.

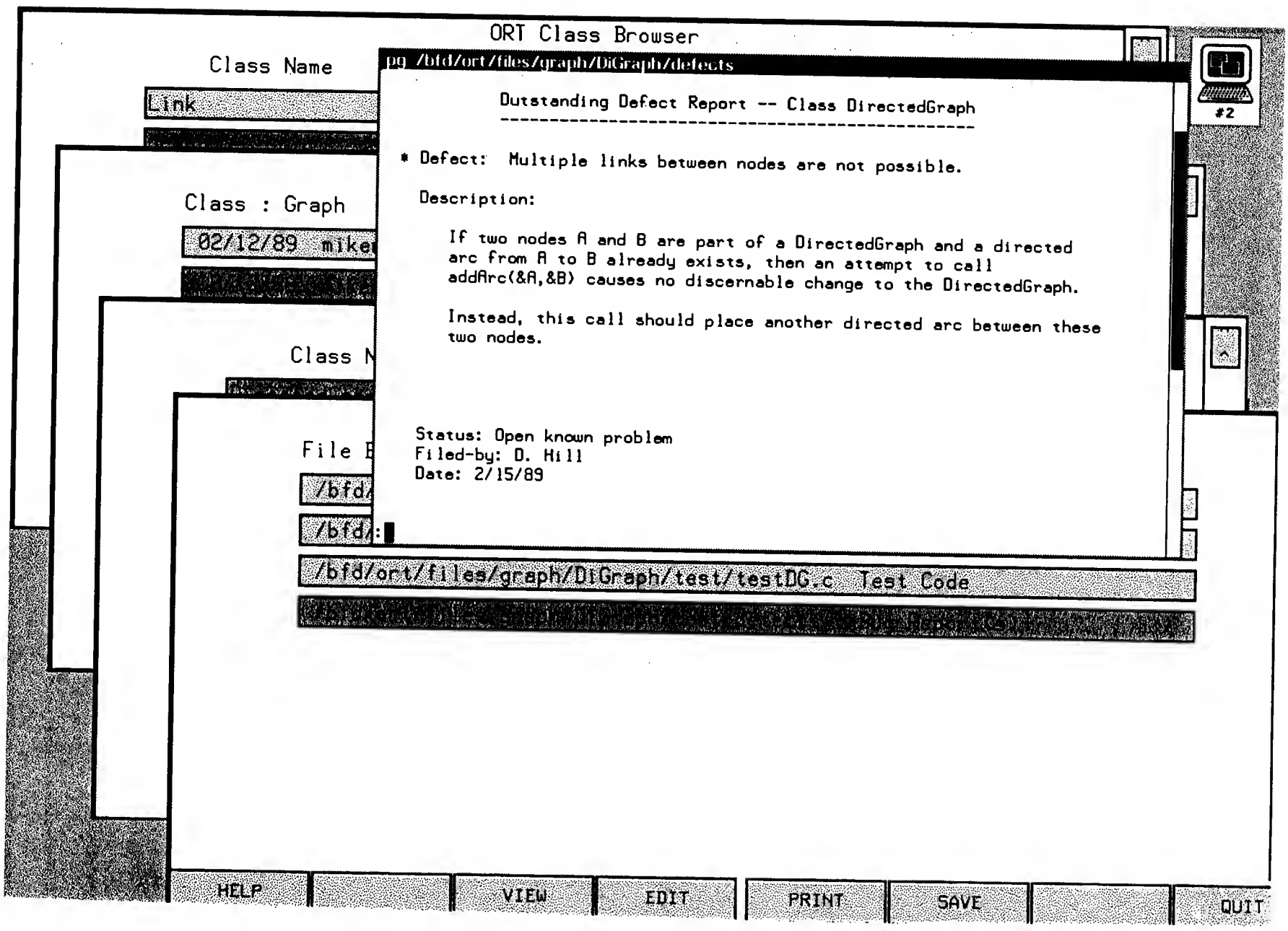

Figure 6-7: The search for "DirectedGraph" (Part 2).

As the user explores the uses and reuses of Graph, two very relevant classes are discovered. The first is DirectedGraph, which is, in fact, exactly what the user thought was appropriate. The user finds this class by selecting the usage attempt mentioning it (see figure 6-6) and, because it inherits from Graph, hits the HEIRS key. This generates another class browser containing only DirectedGraph (the third window from the bottom in Figure 6-7), which the user investigates to assess its reusability. The investigation proceeds positively until the user decides to investigate the defect reports via a file browser (the fourth window from the bottom in Figure 6-7). There, by examining a file containing a defect summary (top window), the user finds that outstanding defects exist for DirectedGraph and that they will most likely need to be fixed, in the user's assessment, before reuse of DirectedGraph can occur.

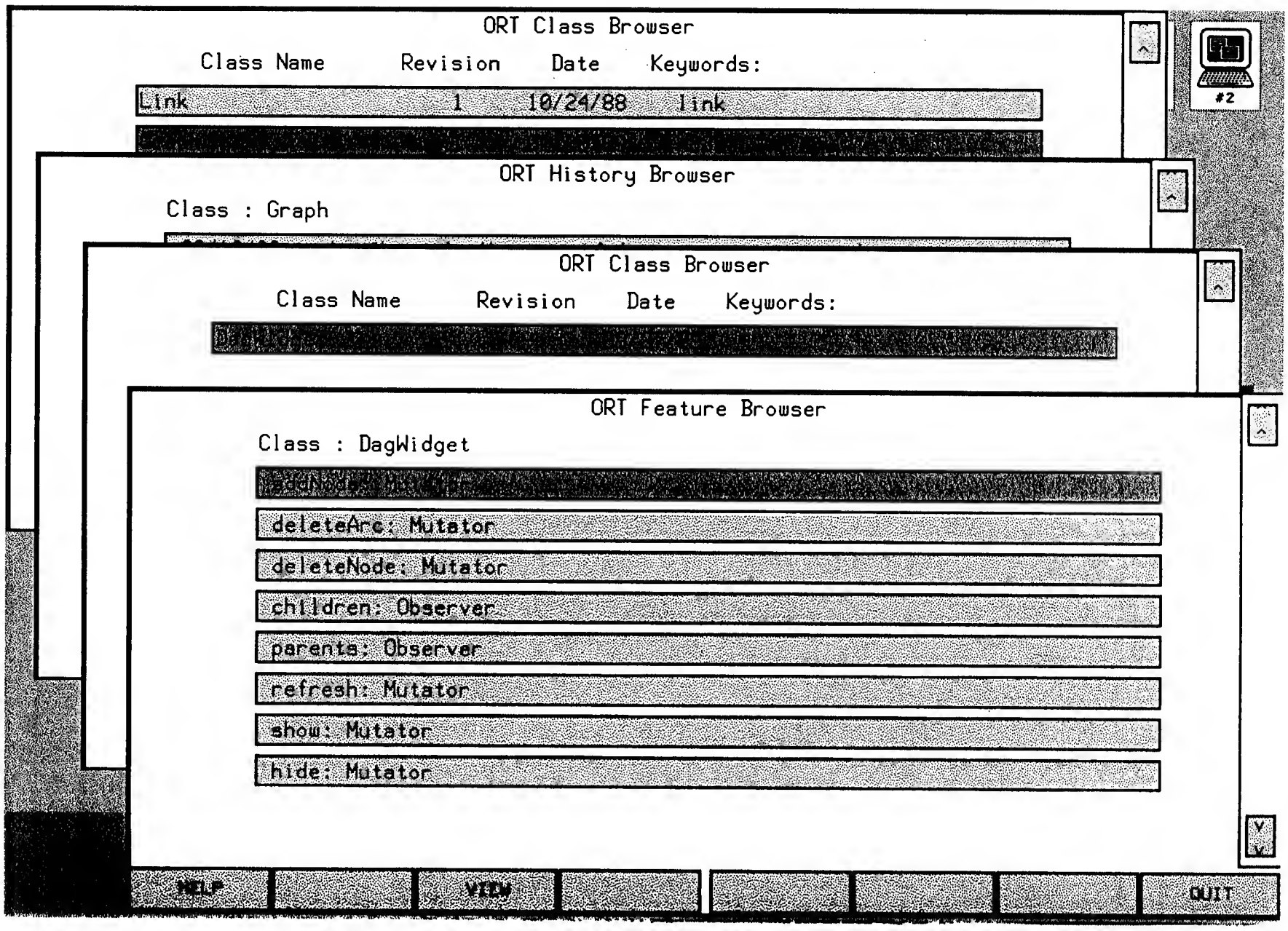

Figure 6-8: The search for "DirectedGraph" (Part 3).

The second relevant class, DagWidget, was discovered as a result of the misgivings the user had about the defects he discovered for class DirectedGraph. The exploration of Directed Graph was backed out to the history browser for class Graph, at which point the user selected a different successful reuse attempt and used the HEIRS key to discover DagWidget, which incorporates even more desired functionality than DirectedGraph. It is a C++ user interface class that inherits from DAG, a directed, acyclic graph abstraction and also displays itself in an X-Window. Figure 6-8 shows the user discovering some of the powerful features of this class in an ORT feature browser.

The user also discovers that DagWidget uses much X-Window-related code to realize its implementation. This is done by generating another class browser with the "DEPENDS ON" key (not shown). Though the use of DagWidget entails the integration of much more code than just the DagWidget class, the desired functionality is met (exceeding previous expectations) with this example of aggregated reuse of a collection of related

classes, both from the graph inheritance hierarchy and the X-Windows user-interface dependency hierarchy.

For brevity, the description of a good deal of browsing has been left out. This browse demonstrates the benefit of *non-local browsing*. "Arc" and DagWidget are related semantically, but are relatively remote from each other with respect to the library. The relationships stored in the library effectively connected them together via a series of links. The user was able to investigate the links leading away from Graph until DagWidget was discovered and subsequently incorporated into the concrete design that follows from ORT's use.

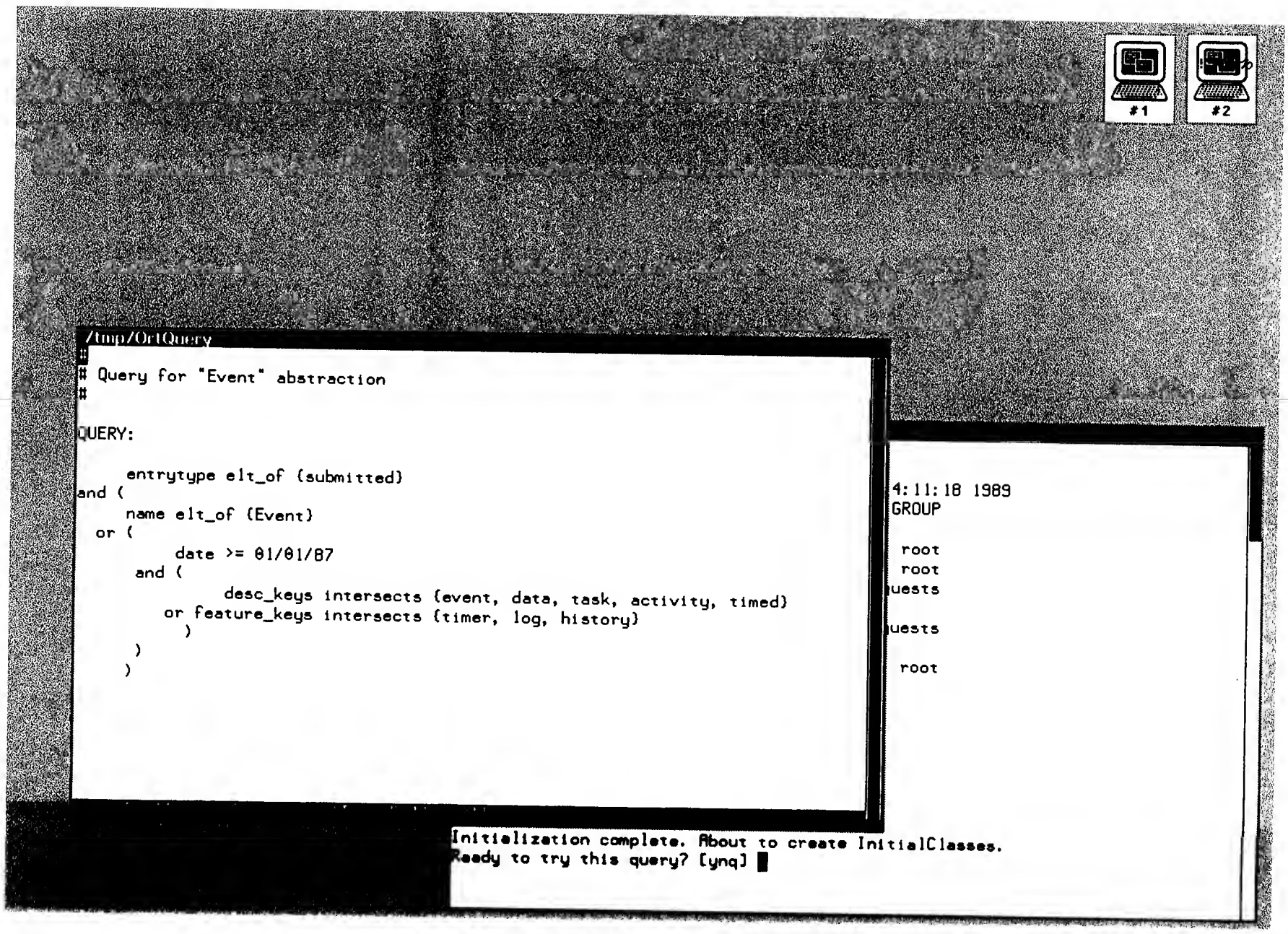

Figure 6-9: User edits query for "Event" abstraction.

The user continues the search for reusable classes by submitting a query for the "Event" abstraction (see Figure 6-9). Since it is unlikely that the particular type of organizational events needed here exist, the user opts to make the query as general as possible. All portability constraints as well as the size and complexity limitations have been removed. The motivation for searching the library in this way is to leverage from past programming experience by finding any examples relating to types of events, the processing of events, call-back mechanisms, and other related programming techniques. With this technique, the user can potentially learn more about how an event-processing system should be implemented. A reusable or leveragable class is sought, but is not expected.

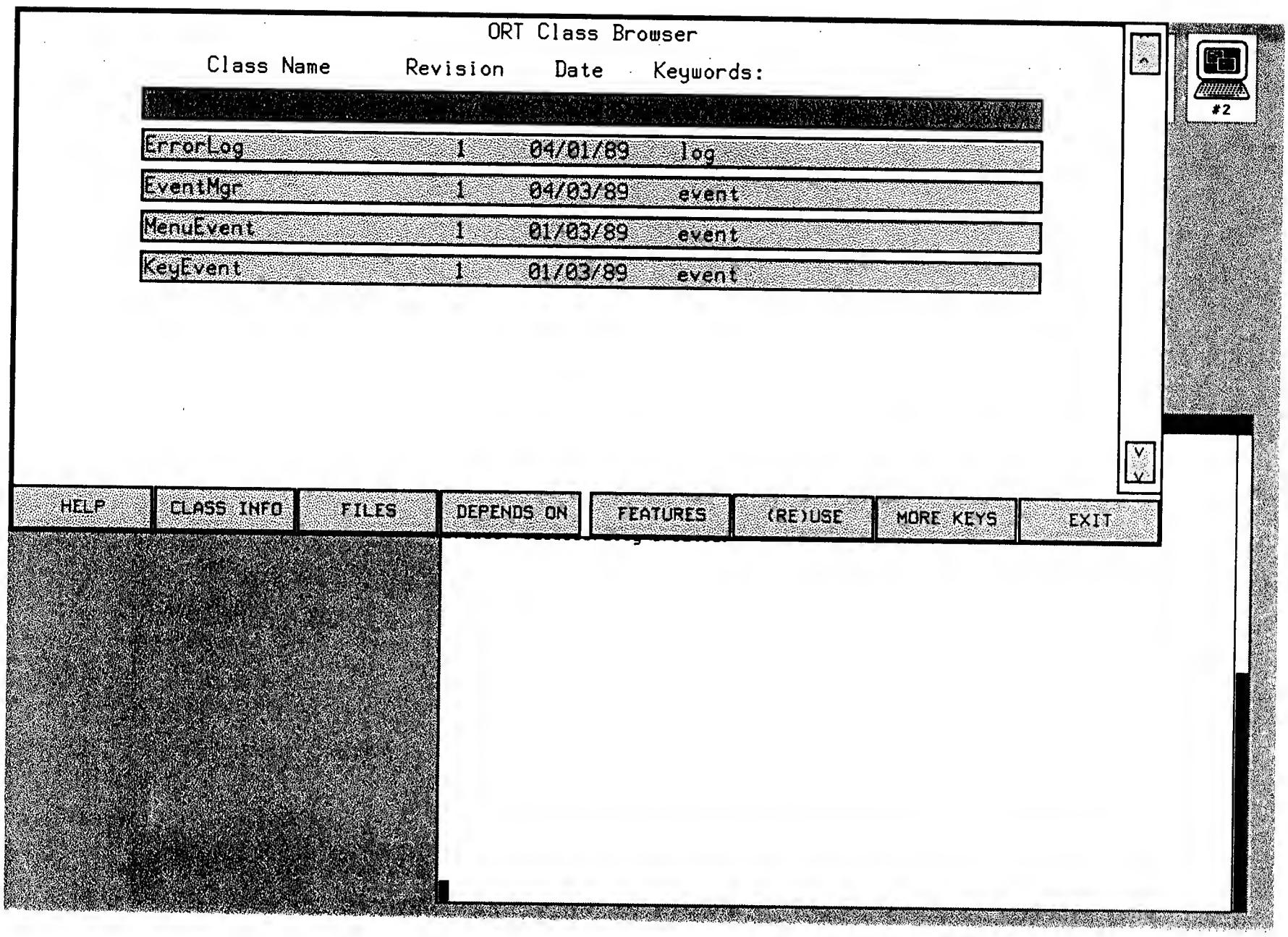

Figure 6-10: Class Browser from "Event" query.

The very lenient query for "Event" does, in fact, produce some relevant results (see Figure 6-10). The primary class browser contains a few promising classes, Timer and ErrorLog, which could definitely be useful in implementing an Event class. These classes were matched because the original description of "Event" suggested that Events have a timer and a log associated with them, which was later expressed in terms of keywords. Only a small part of the exploration of these classes is shown: Figure 6-11 illustrates the effects of hitting FEATURES while the selection in the class browser is for class Timer.

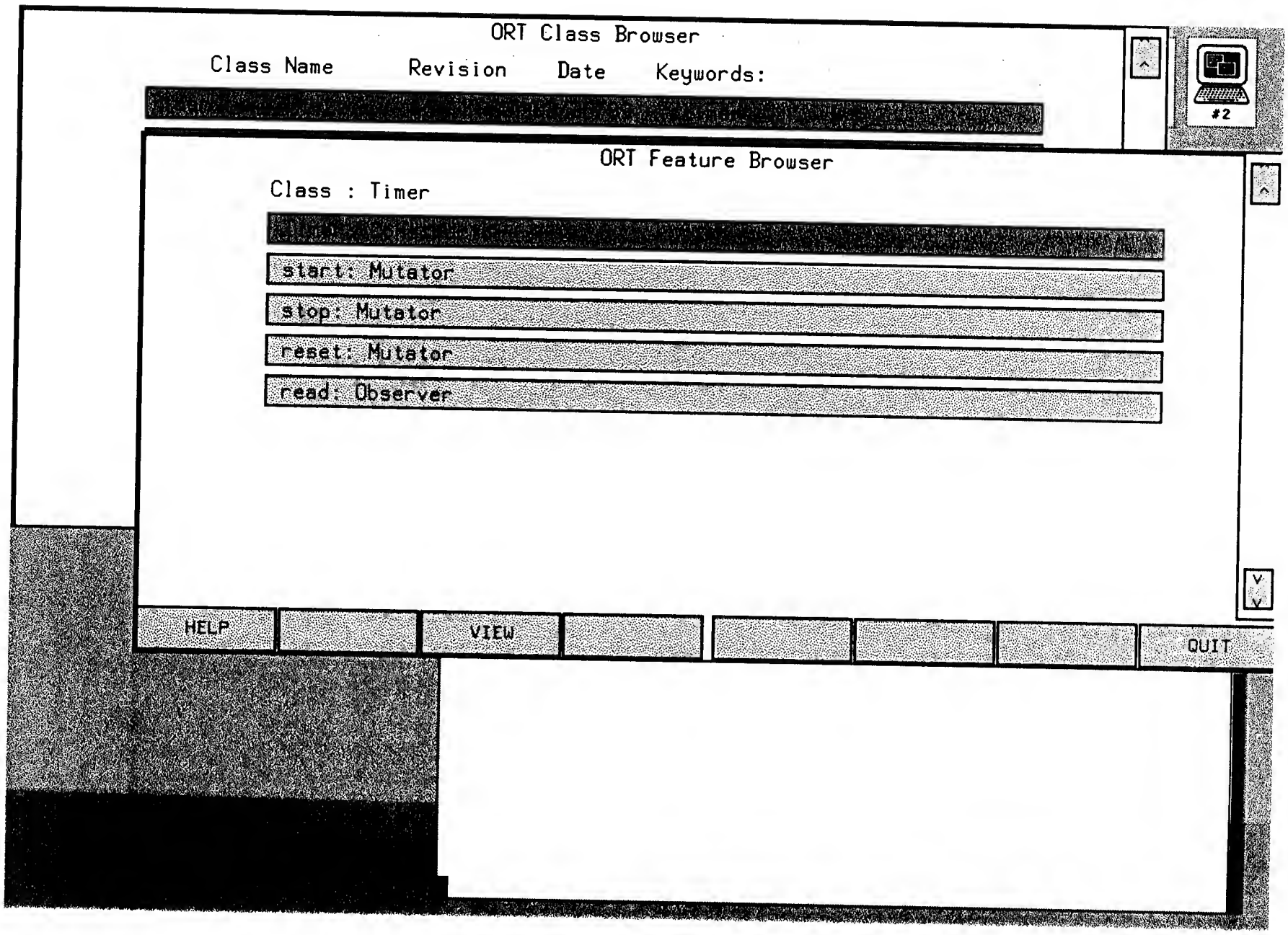

Figure 6-11: Examination of class Timer features.

After backtracking to the primary class browser (see Figure 6-10), the user finds class EventMgr intriguing because it may be a different approach to simulation. Rather than keeping track of a graph with events floating between nodes, an event manager could simulate all the links transparently. However, upon further investigation of the programming techniques used (see Figures 6-12 and 6-13, where the user selects this class, clicks on FILES to obtain a file browser, then actually VIEWS some of the files), class EventMgr was determined to be inappropriate for reuse. Aside from the semantic distance between "Event" and EventMgr, a decision not to reuse can be made based solely on the state of the implementation. Through the direct examination of a header file, the user finds that EventMgr was never truly "factored out" of its original environment, so any attempt to reuse it would entail a major implementation effort.

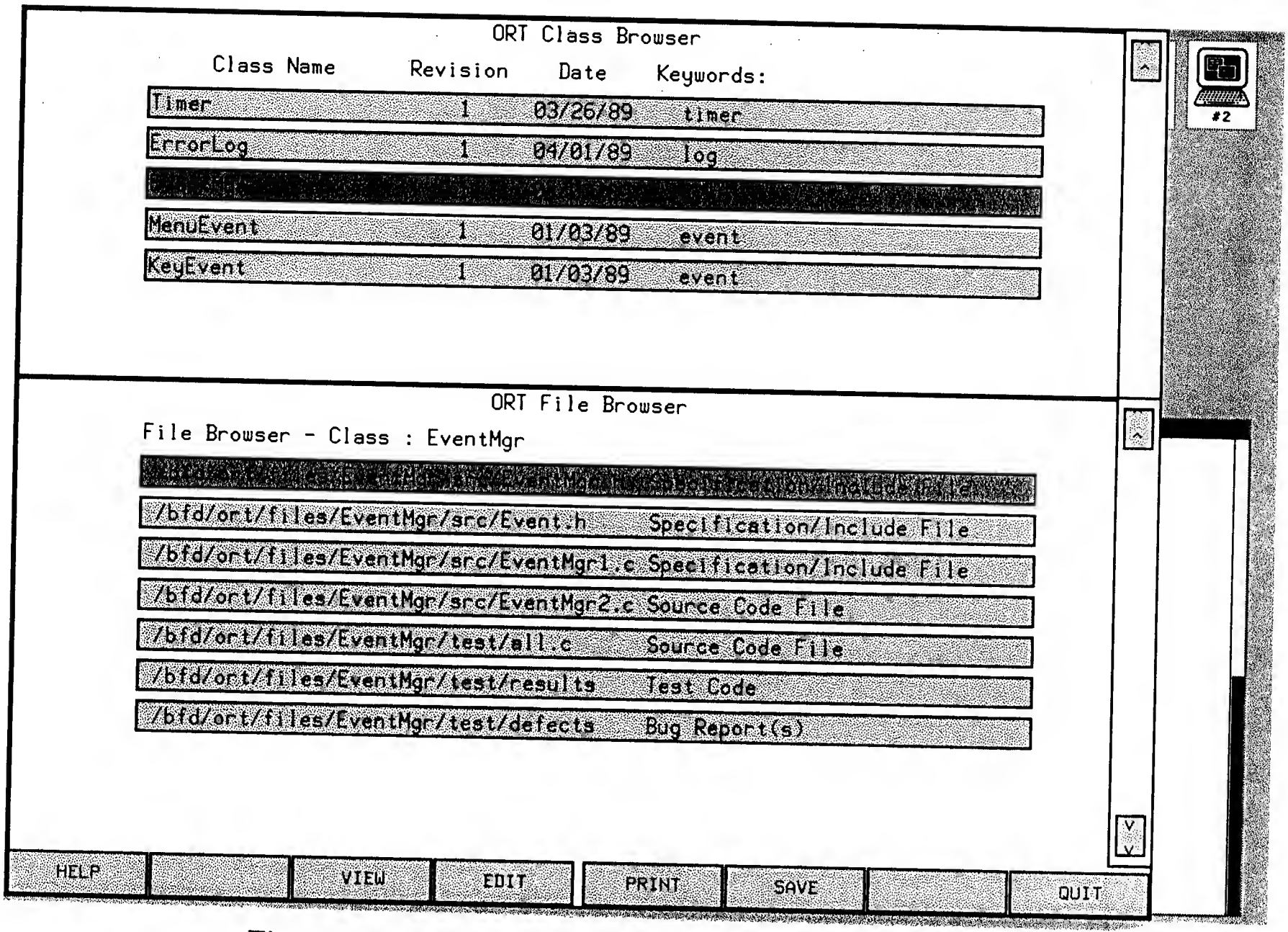

Figure 6-12: Examination of files associated with class EventMgr.

Finally, the user exits and tries one final query. The user-interface portion of the program being designed requires menus, which the user has seen on many projects. The user desires to reuse a specific Menu class which is most likely stored in the ORT library. This class was built as part of a project which is called "Calypso". As shown in Figure 6-14, the user constructs a very harshly-filtering query (most of the previous queries were mild in this respect) to obtain that specific class, if it exists in the library.

After this query is executed, the intended class is found, as is shown in Figure 6-15.

The user ends the session with ORT by obtaining the code for this Menu class. This is done by generating a file browser (shown in Figure 6-16) with the FILES key and then using the SAVE key to save the files to another position in the file-system for integration purposes.

The SAVE key can be configured to copy the file mentioned in the current selection of the file browser to a standard location, specified by the ORTHOME environment variable,

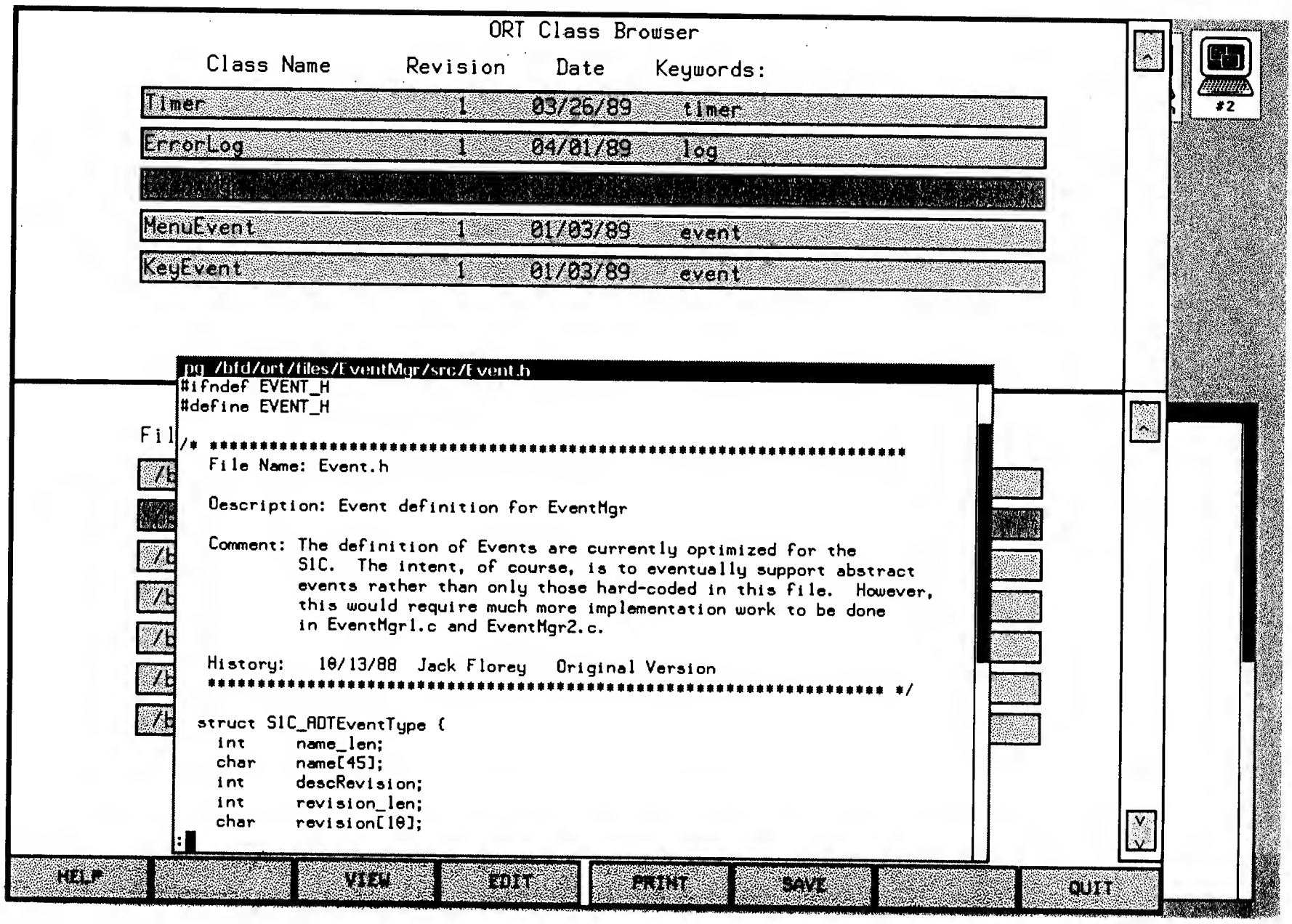

Figure 6-13: Examination of the event definition header file for EventMgr.

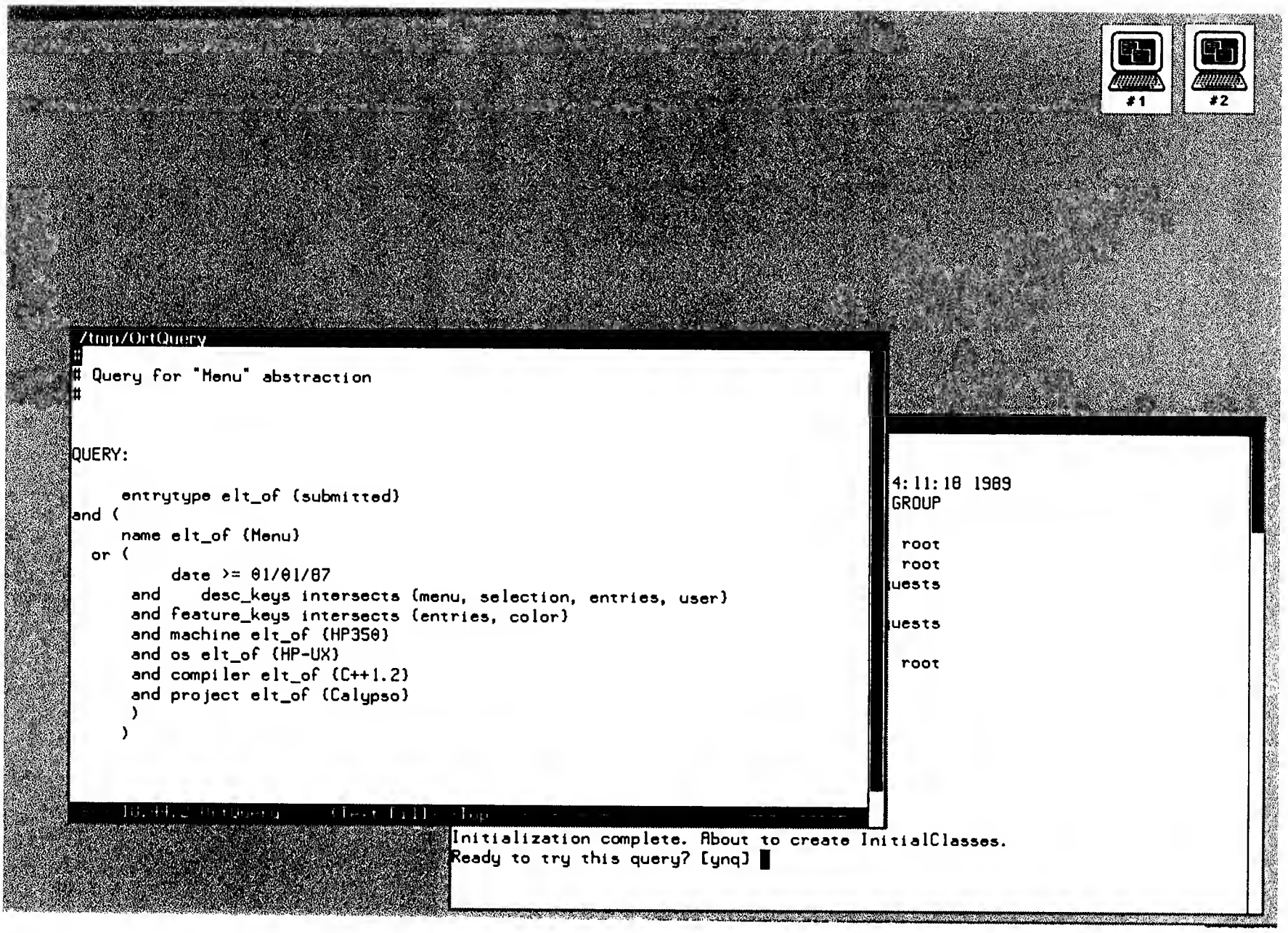

Figure 6-14: User produces a harsh query in search of a specific class.

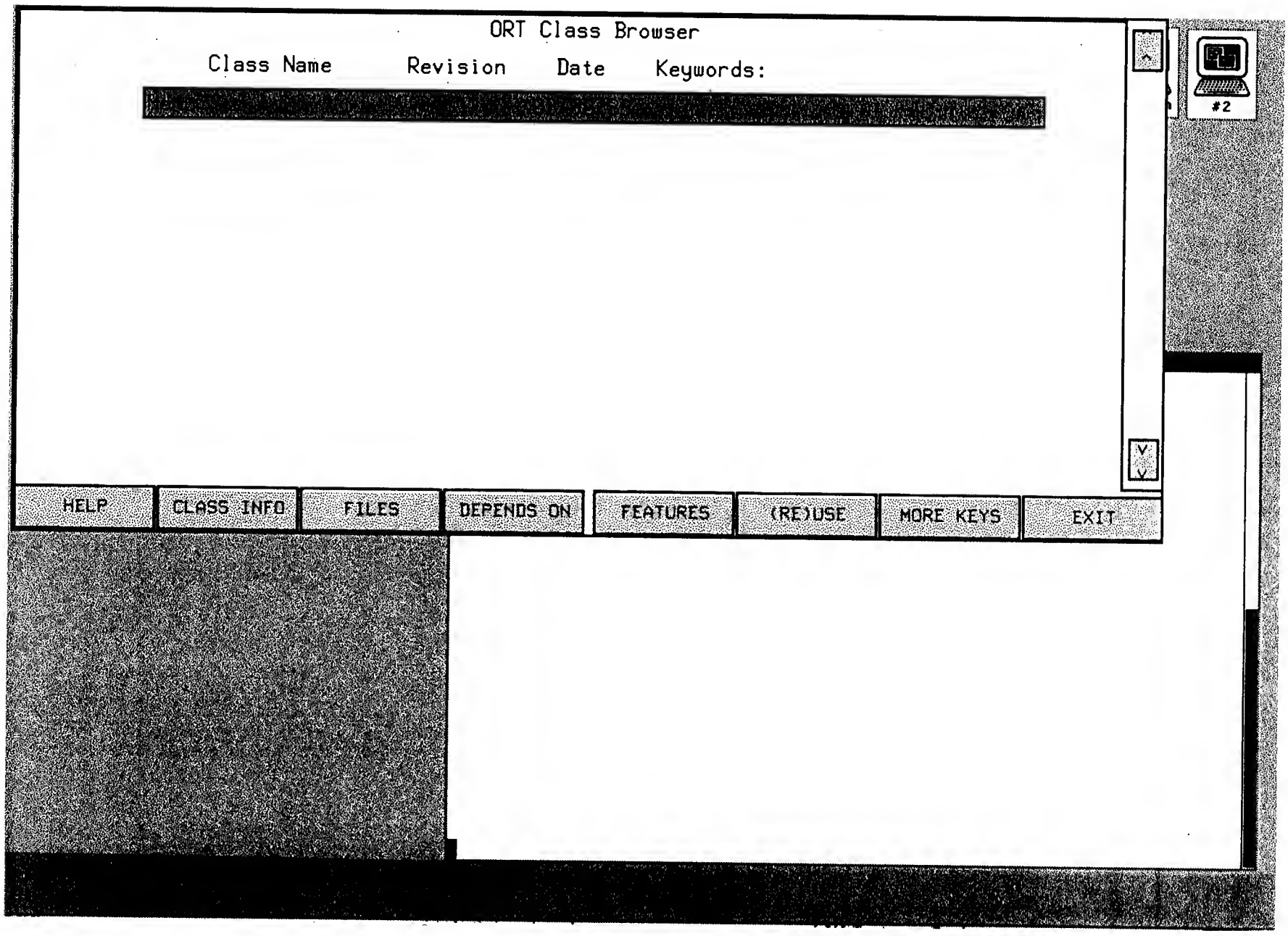

Figure 6-15: Resulting class from the "Menu" query.

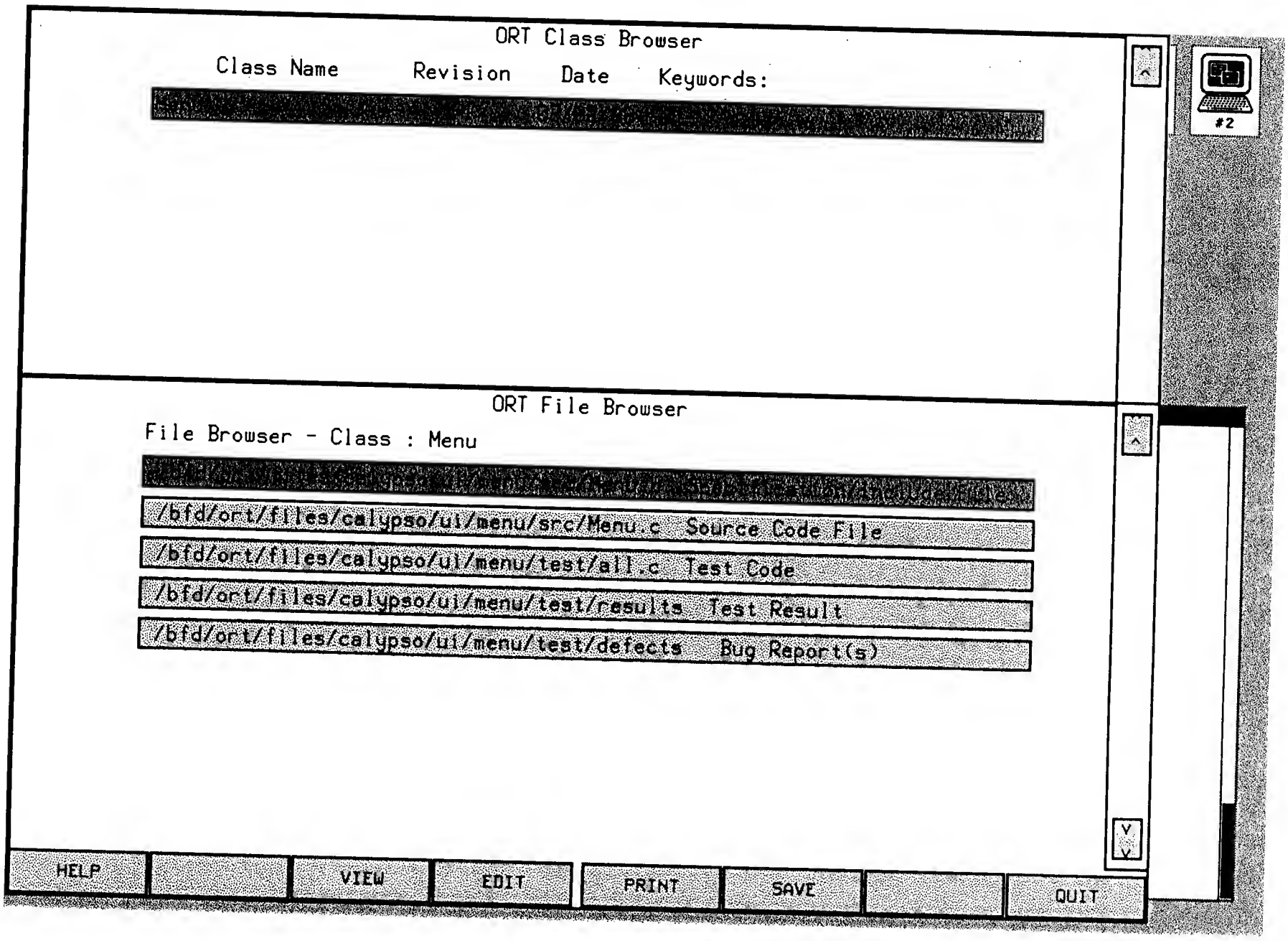

Figure 6-16: Obtaining code for class Menu.

if set by the user. Otherwise, the file is copied to the current working directory of the user where ORT was invoked.

This concludes the scenario demonstrating how ORT can be used to aid in the process of finding the right components to reuse during the early design phases of object-oriented development. The benefits stated at the beginning of the chapter have been demonstrated by specific examples included in the scenario. The essential properties of the ORT query/browse interface have also been demonstrated. At this point, a formal design, including precise class specifications, can be written and implementation efforts can proceed based on the information and files discovered using ORT. Note that the use of ORT requires no specific methodology to perform requirements analysis or develop initial design ideas, nor does it impose upon the programmer additional overhead in implementation save for when it becomes time for Class Descriptions to be entered into the library for the classes developed for the resulting simulation system.

7. Implementation

The implementation of ORT consists of three layers. At the lowest level, a relational database implements a library of Class Descriptions and the storage of other, configurable information. At the intermediate level of abstraction, a set of database interface classes implement a uniform interface for all applications (i.e. tools) needing access to the library. The top level contains the tools, ORT, CHECKIN, and CHECKOUT, that are built on this intermediate level and are user applications of the ORT architecture.

As was described in Chapter 4, the SQL database standard was chosen to implement the bottom layer of the implementation. For this task, HP's version of SQL (HP SQL/HP-UX) was chosen. The database interface classes and applications thereof were written in an extended version of AT&T's C++ [36] developed at the Waltham Division of Hewlett-Packard. Thus, ORT currently runs only on the HP9000/300-series workstations. The tools are designed to work with standard Unix system calls and have traditional Unix command line interfaces. The user-interface of ORT uses X-Windows, because of its high degree of acceptance in today's workstation-based development environments.

7.1 The Underlying Database Definition

The library itself, in its most concise form, is expressed in terms of the entity-attribute-relationship model (see Figure 4-1). This model was translated into an equivalent relational form expressed in Structured Query Language. The resulting SQL database definition has also been augmented by a number of tables and declarations in order to implement appropriate functionality at this bottom-most level.

7.1.1 The Translation Process

Little research has been done in the area of automating the translation process from entity-relationship models to relational models. Methodologies, however, do exist for translation in the opposite direction [7, 25]. There are some more-relevant methodologies [3, 8] as well. The following procedure outlines the manual translation process used to obtain SQL code for the library of Class Descriptions:

1. For each entity in the model, a relation is created containing only an integer identifier. This index is used to access entities of this type by other entities in the system. The resulting table definition of this relation is called an *entity table.*

2. Attributes of the corresponding entity are added to the entity table and are thus assigned a data type (like INTEGER or VARCHAR(40)). In cases where an attribute is a text string of arbitrary length (e.g., the long class description), a separate table

is defined containing three columns: (1) the text data field, defined to be as long as permitted by the database implementation, (2) an integer data field used to sequence "chunks" of text in the first field up to the amount required, and (3) an integer field to link a set of entries back to the entity table from which it came.

3. Relationships between entities are implemented in one of two ways:

   (a) If the relationship is of type "is_a", then the two entities can be combined, if deemed appropriate. For entities A, B, and C, if A "is_a" C and B "is_a" C then a union entity, C', can be defined so that, when translated to a table as described in steps 1 and 2, the resulting table can be indexed and searched efficiently on attributes common to both A and B (i.e. those of C).

   (b) In all other cases, the integer identifier column added in step 1 is used by creating the link as a column of the "originating entity" for the relationship. This column is of type integer and contains, essentially, pointers into the other table. For example, nearly all the tables are indexed on a single column, an integer data field *classID*. The *classID* is unique for each Class Description and serves to link many of the entities together, as do the links pointing away from the class entity in Figure 4-1.

Finally, a "hack" was added to the resulting system to provide version control. Since it is conceivable that an administrator of the library may want to use a previous version of a Class Description, multiple versions are kept in the database and old versions must be manually purged. The addition of one column to the Class entity table allows a "description revision" to be kept. Uniqueness of a Class Description is therefore defined by a triple, containing the class name, the implementation revision of the class, and the Class Description revision.

7.1.2 Configurable Information in the Database

Additional tables were added for storage of configurable information. ORT accesses these tables rather than using "hard-coded" constants. This also allows faster comparisons when searching for matching components, since, in all cases, a string is resolved to an integer identifier (ID).

- Feature Types – A one-to-one mapping from a string to integer ID, enumerating the types of features a class can have.

- File Types – A one-to-one mapping from a string to integer ID, enumerating the types of features a class can have.

- Comment Types – A one-to-one mapping from a string to integer ID, enumerating the types of comments that can and should be made about a class.

- Search Field ID's – A one-to-one mapping from integer ID to a table name and column name in the database.

Further, the following set of tables also helps to map many different textual expressions which mean the same thing to one identifier.

- Operating System ID's – A set of many-to-one mappings from strings to integer ID for efficiency in comparing OS dependencies.
- Machine ID's – A set of many-to-one mappings from strings to integer ID for efficiency in comparing hardware dependencies.
- Compiler ID's – A set of many-to-one mappings from strings to integer ID for efficiency in comparing language dependencies.
- Project ID's – A set of many-to-one mappings from strings to integer ID for efficiency in comparing project or environment dependencies.

7.1.3 The SQL Definition

A shell script was developed for the SQL definition that defines all the tables associated with the entity-attribute-relationship model of the Class Description. This script generates the database and initializes it. Also, a number of other things are accomplished by this process:

1. **Multi-User Access:** The database is configured to allow multiple concurrent transactions, meaning that a number of users can run ORT in parallel. Locking on write is therefore implemented at this level as well.
2. **Indexing:** To speed database transactions, integer ID columns of the entity tables are made SQL indexes for that table so that links between entities can be followed quickly.
3. **Security:** The access-control list for modifying the ORT library is implemented at the SQL level by allowing anyone on a list of trusted individuals to modify the database. Should one of the tools be run by someone not on this list, SQL will cause a database write error.

7.2 The Database Interface Classes

The data in the Class Description are broken down into *fields*, or chunks of closely-related data. There is a base class Field which contains the generic capability to access the database using dynamically-prepared database commands. Class Field is also a template for the actual field classes, each of which knows about only a portion of the database and can manipulate that data in a variety of generic ways, as defined by class Field. This architecture serves the following purposes:

1. Each type of Field object can:

   (a) Store its data to the database for a specified Class Description.
   (b) Retrieve its data from the database for a specified Class Description.
   (c) Display this data (useful for information not presented in a browser).
   (d) Unparse this data.
   (e) Parse this data.
   (f) Format this data for hardcopy output (see Section 8.1.1 for more details).

2. The layering of this set of interface classes over the database hides details and makes implementation of the tools much less tedious.

3. This set of classes provides an object-oriented veneer over the underlying relational database implementation.

4. Field classes can encode information present in the entity-relationship model that is not easily expressed in SQL. For example, relationships are stored as integers; the knowledge about what table this integer indexes into is more easily kept in the field class.

5. When using this interface, the granularity of the operations on the contents of the database is correct for the model ORT requires. For example, a Class Information viewing window calls the display function on each of the fields not already capable of being presented by some browser. Thus, transactions can be made at high-levels, but in a selective manner.

6. The information stored in the additional tables (described in section 7.1.2) can be automatically loaded into memory as part of the creation of a Field object. This allows subsequent use of the information to occur at the same speed as if it were hard-coded.

Developing database Field classes is more difficult than using a true Fourth-Generation Language (4GL) tool to manage the necessary relational database interactions. However, a serious loss in flexibility must be taken with many currently-available 4GL's. Such tools will become increasingly sophisticated, as will entity-relationship and object-oriented databases. A conversion to one of these platforms would be appropriate at a later date.

On top of the collection of classes that inherit from class Field, there is class ClassDesc, which is immediately called by the applications. This class is designed to model a Class Description; it allows operations to be performed across collections of Fields representing the contents of a Class Description. In the abstract, a ClassDesc object contains all the data of a Class Description. What actually happens, however, is that operations on a ClassDesc generally cause database operations.

7.3 The Tools

The prototype tools that were implemented are ORT, CHECKIN, and CHECKOUT.

7.3.1 ORT

ORT provides the query/browse user interface and, as such, most of its implementation is contained in the compiler for language *Q* and in a collection of browser classes.

The *Q* compiler is a stand-alone program which ORT runs after an input file has been prepared. This input file is prepared by creating a temporary file containing a query template and allowing the user to customize it. This query template comes from the users .ort file, if it exists, or the system default file otherwise. Once this temporary file exists, the user is placed in his or her editor (i.e. EDITOR environment variable) and asked to prepare the file. Upon completion, the resulting query is run through the *Q* compiler.

The implementation of the *Q* compiler consists primarily of an input file for lex and an input file for yacc. Save for the initialization procedure and the error-reporting procedure, the entire compiler is generated by these two Unix system tools. A textual query is built up for output based on the query input by the user. Should an error be detected by the compiler or in the output of the compiler, a message is generated and the user's editor is re-run.

The compiler output is part of an SQL query, which ORT then completes and executes. ORT provides the portion defining what data is accessed, where it is retrieved to, and the order in which it is retrieved. The remainder, provided in the compiler output language, defines what tables are accessed, how they are joined for access, and what conditions must be met for the select clause. These select conditions, because they are translated from the original *Q* expression, can be of arbitrary length (ranging from as little as a hundred characters to as much as a few thousand characters).

The list of classID's resulting from the execution of this dynamically-prepared SQL query is then used to create the primary class browser. Once created, the remainder of the execution of ORT consists of "running" this browser and destroying it after the user initiates its termination. The other activities relating to a browser, like maintaining and changing the current selection, paging entries after scroll-bar event, and executing soft key commands, are handled internally while the *run* member function (a function feature in C++) is executed.

Soft key commands can generate other browsers, which are "run" from the primary browser. Each browser is spawned as a light-weight process (or thread) and runs independently until it is no longer desired by the user. The soft key commands also provide for regenerating another primary class browser during the browse. Thus, the RE-QUERY key destroys the primary browser and runs the user's editor on the temporary file used as the previous *Q* input. The primary browser is then regenerated from a new, further-modified query and browsing continues with the new browser as well as any existing browsers derived from the old primary browser.

To summarize the classes used to implement the browsing functionality, there are essentially three levels of browsers in ORT. The lowest level consists of the Browser Widget, which is the graphical object containing a menu, a scroll-bar, and a set of eight buttons (or soft keys). A Browser Widget maintains the current selection by reacting to mouse-clicks on the menu entries. It also pages entries into the menu and highlights the current selection after a scroll-bar event. A Browser is generated by first creating a BrowserParm (browser parameter) object. A browser parameter contains the configurable traits of a Browser such as: arrangement and size of keys and menu entries, color scheme, and size and position of the browser window on the display.

The intermediate level of the ORT user-interface consists of the ORT_Browser, which contains a graphical Browser object as well as the implementation of the functionality needed to process soft key commands for ORT. The processKey member function is set up as a call-back for soft key activations, from which the appropriate internal function is dispatched when a key is depressed. A current KeySet, containing key labels, help text, and key codes for a set of eight soft keys, is contained by the ORT_Browser. The context-sensitive help facility is provided by the ORT_Browser as well. The top level is a collection of specializations of this general ORT_Browser. Inheriting from this class are the classes ClassBrowser, FeatureBrowser, HistoryBrowser, and FileBrowser, which have all been described previously.

7.3.2 CHECKIN and CHECKOUT

CHECKIN and CHECKOUT perform related operations. In the first case, a ClassDesc object is created using a ClassDesc constructor that takes a file pointer and parses from it to create a new ClassDesc, should the parse succeed. This constructor parses by mapping the "parse" member function across a list of Field objects which it maintains as the representation of flat-file format. Should any part of the parse fail, there can be two courses of action taken, depending on how the program is run.

CHECKIN, when run from the command line, takes an optional file name argument. If this argument is provided, then it is considered to be run interactively. Otherwise, the input is assummed to come from standard input (so as to integrate with other tools which might add further automation to loading the library). If the tool is being run interactively, then the user is prompted to re-edit the file. In either case, an appropriate error message is generated, along with a line number that locates the perceived error in the file or input stream.

CHECKIN checks the "checked out" data field in the Class entity table to determine if the parse function should be called. If it is different than the user's login ID, then an error is generated. If it is blank, the user is prompted for assurance that the data should be updated. If the class is new (i.e. no class by that name and revision), then a new chain of Class Descriptions is started for it.

CHECKOUT also creates a ClassDesc object as well. It uses another constructor used to access the database and find a ClassDesc with a given name and revision (provided on the command line), if it exists. The newest version of the Class Description is returned,

which is then unparsed onto standard output. Command-line options control whether a lock is attempted on the Class Description. If a lock is attempted and the Class Description is not already checked out, then the "checked out" and "checked out time" data fields are set to the user's login ID and the current time.

CHECKIN and CHECKOUT use the ClassDesc member functions *parse* and *unparse*, respectively. These call the *parse* and *unparse* member functions on a list of Field objects, flatFileList, which denotes the order of the fields in the flat-file representation of a Class Description. Each Field object, although using as much common Field functionality as possible, re-implements the *parse* and *unparse* member functions to recognize and format, respectively, the data for which they are responsible. Because of the simple format used in the flat-file representation, the parsing is done via recursive-decent techniques and the unparsing is done merely by formatting information so that it can be recognized by the associated parse function.

8. Conclusions

There are many things left to be done before ORT can be effectively used in an actual software development environment. Most of this effort involves further development upon the architecture developed for the purpose of prototyping ORT. Additional research could add further value to ORT by embedding in it knowledge about, for example, class composition. Despite the logical extensions that can be made to this work, there are limitations inherent in the ORT approach. Conversely, there are benefits unique to the approach as well.

8.1 Future Development

The following collection of extensions to the current implementation would round out and complete a useful system for facilitating reuse in a modern object-oriented software development environment.

8.1.1 Hard Copy Documentation Generation

Currently, much of the documentation for classes is contained only in files which are to be the input to some formatter, like LaTeX or troff. If ORT were in use, however, this would require documentation to be entered directly into the library as well.

There are certain undeniable benefits of having physical hard copy references to consult, especially when they can be formatted in pleasant ways using existing technologies. These benefits include: portability, ease of marking locations and adding information via notes, highlighting, and "Post-its", increased readability, high availability, and many others. Certainly, it would not be appropriate to force software engineers to give up hard copy documentation of these classes.

There is an alternative to requiring two different sets of documentation that must be updated and kept consistent. CHECKOUT and the database field classes have already been augmented with the appropriate structure to generate such formatter input-files, although no specific functionality has been delivered. The assumption is that the information in a Class Description can be, essentially, prepared adequately for the formatter in an automated fashion. The remaining work is to populate code into the heirs of class Field in order to allow a different type of output for the formatter of choice. This function could then be invoked by another tool, or by a command line switch to CHECKOUT.

8.1.2 Automated Data-Entry Tools

The information in the ORT library must be entered manually, at considerable expense to the software development process. Much of this information can be directly extracted

from the implementation of a class. An automated entry tool could derive from the code a set of database primitives which, when executed, would install this information into the library, or update it after a class has been modified. This tool, though not implemented, is based on known compiler and database technologies and would therefore not be difficult to realize.

The acquisition of the data that can be derived from the implementation eliminates the tedious entry of name, date, contact/author, code metrics, feature definitions (the class interface), and other assorted pieces of information, dictated by the sophistication of the tool and its use. The remainder of the information would later be entered using the CHECKIN and CHECKOUT tools.

A reasonable implementation of this automated data-entry tool would be contingent upon the ability to reuse the front end of the language compiler for use in parsing and lexing the implementation files. Essentially, one would build a compiler whose output is database storage rather than object code.

8.1.3 Direct Library Access/Update Tools

A set of specialized tools that can be used for quick, one-time access or update of information in the database would be of great practical value. It is anticipated that certain capabilities, although already provided by ORT and the library-editing tools, ought to be available as stand-alone functionality. These small sub-tools would help avoid the sometimes cumbersome nature of the current ORT in cases where only a small amount of information is modified or extracted.

Two examples of such library-accessing tools are (1) a tool that, provided with a class name and revision, returns a list of all classes on which the implementation of that class depends and (2) a tool which can accompany the use of CHECKIN, providing support for finding synonyms (based on an on-line thesaurus) for keywords for the classification of a class.

Two examples of library-modifying tools are those which allow direct entry of comments or reviews for a class. Should a programmer arrive at significant realization about a class, it should be made easy for him or her to add a comment to the library and review what others have said about it as well. Reviews should be made by a set of individuals skilled in either the domain of the class or in programming in general. It may be appropriate to implement a review-by-electronic-mail system to insure that reviews are accomplished or it may also be appropriate to be able to directly add a review to an existing Class Description.

8.1.4 Library Summary Tools

It is anticipated that the query/browse tool may not be sufficient to quickly convey overall information relative to class definitions and interrelations. Supplemental tools that generate graphical displays, plots, or print-outs would be beneficial in getting views

for use when a larger-scope picture is desired. Certainly, both an inheritance and dependency hierarchy display tool would be useful to graphically depict the architecture of projects built on this object-oriented paradigm. It is also reasonable to desire other summary-based tools useful for a variety of purposes, like project management summaries, for example. There are many such tools in existence that could be ported to the ORT framework.

8.1.5 Syntax-Directed Editing of Flat-Files and Queries

The editing that ORT and CHECKIN require are heavily laden with syntactic structure. The flat-file format is essentially a form that could be more effectively filled out using an editor that advanced the cursor to the next field (in the style of a block-mode terminal-based application). Some text fields can be added to ad infinitum, so this technique must be augmented by the concept of textual data fields without bounds.

Similarly, a *Q* expression mode for the user's editor would afford the ability to catch syntactic errors without having to run the compiler, just as syntactic knowledge of programming languages allows editors to assist code production.

8.2 Further Research

There are a number of possible advances that require further research. The classification scheme of any reuse library is crucial to its success. Though automation can do little to increase the quality of the collections of keywords associated with Class Descriptions, it can assist in the process of choosing keywords for a class. The other field of additional research strives towards the goal of software synthesis by suggesting ways in which to combine library components to achieve the desired component.

8.2.1 Assistance in Classification

ORT's classification system relies on the proper choice of keywords in the classification of components and in querying for them. Currently, keywords must be chosen and entered manually as part of a Class Description. Keywords are intended to be derived from textual fields in the Class Description. A logical extension would be to assist the user by running this text through a filter that removes all trivial (like "the" and "for") and ambiguous (like "output" and "represent") words and possibly performs some substitutions based on a thesaurus of terms for classifying classes. The user could then edit this collection of keywords before finally submitting the Class Description.

Additional keywords could be added during this process that are synonyms of existing keywords. The resulting list would then need to be pruned by an individual or individuals with some knowledge of library science, the domain of the class, and programming in general.

8.2.2 Desired Class Descriptions

One step towards synthesizing desired components is to effectively describe the desired components. If the representation of a query were essentially the same as the representation of a Class Description, then users would be encouraged to think about and specify the many properties they desire in a class rather than submitting a more lenient specification and then relying on browsing. This approach is particularly effective in cases where exact matching is desired. For example, features can be defined to match when the data type and formal parameter data types match the user's specification of features.

If ORT were to incorporate functionality to suggest class combinations as approximations of goals, more details about the user's goal would help as well. By providing the capability of matching a Class Description of type "query" (a new entry type), ORT would effectively make the Class Description format be the formal class design result sought by designers so that ORT can be used to approximate realizations of them.

8.2.3 Automated Composition

The composition activities of an ORT user can be expedited by suggesting some combinations of entries that approximate the description provided by the user. Although adding this functionality would first entail research into the nature of programming-in-the-large and how programmers glue chunks of programming knowledge together to make systems, ORT provides a platform upon which automated object-oriented composition algorithms can be prototyped.

Part of proper software development is to choose the correct parts when reusing individual modules. If chosen correctly, a single combination of objects provides, in most cases, more versatility and/or less development time than any other possible combination that could be used. Although the guidelines used by programmers to choose certain combinations over others is not well understood, finding many relevant possibilities to choose from could help this process, especially if it can occur in an automated fashion, relieving the programmer of tedium.

Unlike other methods, object-oriented methodologies provide their own composition technology. The composition and decomposition of modules occurs in a controlled fashion. This control is provided by the semantics imposed by object-oriented programming languages.

Object-oriented design can be viewed as a *software decomposition technique*; it bases the modular decomposition of a system on the relatively stable objects the system manipulates, not the functions the system performs, which may change in time and may not be well-defined at project inception [32]. There may be methods whereby classes can be derived from a combination of lower-level classes in a controlled manner, thus reversing the decomposition process to some extent.

The three combinational techniques in object-oriented programming are:

1. Inheritance.

2. Composition (i.e. filling slots in parameterized classes).

3. Containing objects as attributes (i.e. message passing or function call).

Formal understandings of each of these techniques may afford the ability to add value to ORT via suggestions for possible combinations.

8.3 Related Work

A number of library-based systems have been implemented over the past four decades. Recently, there have even been attempts to combine software library technology with object-oriented technology (or at least abstract data types). These attempts generally differ in three important ways from the approach described here:

1. Few provide direct support for those advantages that can be derived from the use of an object-oriented programming methodology. For example, effort has been spent recently on creating libraries of Ada modules. However, Ada is not an object-oriented language. It offers only the capability to program with abstract data types.

2. Little attempt has been made to collect, organize, and present the additional information (i.e. other than code) associated with the software development process that ORT uses to help support the software reuse process.

3. Generally, only one half of the query/browse interface is provided to locate information in the database: either exclusively querying or exclusively browsing.

The work most closely related to ORT is being done by Arapis and Kappel [2] at the University of Geneva. Their approach of an Object Software Base (OSB) is similar to that of ORT in many respects: language-independent storage of class representations in a semantic-network used for future browsing. However, ORT differs in a two key ways.

First, the ORT library design differs from that of the OSB. ORT treats the classification of entries in a more flexible, more finely-grained fashion. The multi-faceted classification scheme in ORT is rather different from the large-grained categories used to group objects in the OSB. The OSB scheme also makes no attempt to record the additional information in ORT like the history of usage attempts on an entry or comments made by software engineers about the class.

Second, the interface to the potential reuser is quite different in the OSB. The interface of the OSB is based on the concepts of "working space" and "object class of interest." Rather than providing a querying interface, as ORT does, the user must decide which working space is appropriate and then move from class to class within this collection to find something of interest. In ORT, relationships between classes become links to follow in the browse, which is begun much closer to the desired entries due to the initial querying.

The Reusable Software Library (RSL) [14], is built primarily for the reuse of Ada packages and descriptions thereof in an Ada Program Design Language (PDL). A collection of software attributes has been formalized to help in selection by querying. In addition, a process involving entry-by-librarian and quality-assurance validation is at least designed. Their plan is to eventually interface with a separate configuration management program to obtain history and usage information, but this functionality is currently lacking, as is the collection of comments on the database entries. The classification system is useful for effective querying based on a "category code", but no browsing capabilities are planned.

A database-storage system for object-based programming was designed by Beach and others at HP-Laboratories [4]. Their object "Protocol Browser" makes a strong distinction between the abstract protocol, or external interface, and the class implementation. It manipulates code and class protocols only and is used for program development and consistency checking. Browsing is supported, but querying is not. Unfortunately, the object-oriented database used to implement this browser poses serious performance problems upon its practical usefulness.

The CATALOG Information Retrieval System [20] used within AT&T is a general purpose database tool that has been adapted for software reuse. No object-oriented approach is taken, nor is there much information to supplement the code. However, there is a great deal of flexibility in the system, which offers a searching capability based on querying using keywords and syntactic variations (e.g., sort, sorting, sorts). A module prologue (or description) template has been developed to describe software components, but little tool support has been implemented to enforce and/or automate the entry of this information.

The classification and search mechanism built by Prieto-Diaz [34] provides a general solution to the problem of finding reusable software components in a large collection. However, no support for browsing is provided here either. Additionally, no support for object-oriented software development is provided. There are a number of other similar attempts at this sort of traditional software library system as well.

A few other systems have been built that rely on an enclosing custom environment. For example, Kaiser and Garlan's work [28] relies upon the fact that all software is written in Meld, their custom object-oriented language. Some functionality in this area is also planned in Bertrand Meyer's Eiffel programming environment [33]. Additionally, the Demeter system [30], the MOMO environment [16], and the Kaiser/Feiler architecture based on SMILE [27] provide some integrated solutions in this area. ORT is a standalone system and makes no assumption about the language or environment used.

8.4 Limitations of the ORT Approach

More specifically, there are inherent limitations to ORT that can not be avoided by merely continuing development and research along these lines. Such deficiencies are pointed out below:

- ORT helps very little without the aid of an editorial committee to monitor the libraries contents or actually take over the job of (re)writing Class Descriptions.

- As does most any attempt involving software libraries, ORT still requires the same or more effort to program things the first time. In fact, more up-front effort should usually be placed on software development techniques so that reuse can begin to occur. More effort should also be expended to continually increase a component's reusability, since software is seldom, if ever, perfect.

- The keyword-indexing approach of ORT is limited by the lack of constraints on the vocabulary. Even if an accompanying thesaurus were added to ORT, there will always be keyword matches that should have happened and did not, as well as keyword matches that happened and should not have.

- The definition of the Class Description is not entirely suited for non-object-oriented software development. In reference to Figure 4-1, it can be seen that many parts of the Class Description do not apply to (but do not interfere with) the use of ORT to facilitate the reuse of traditional procedures and functions. These are the "Generic Parameter" entity, the "Data" entity, and the "inherited_from", "has_parameter", and "has_heir" relationships. A function or procedure could be emulated by a Class and a single Function Feature entity. Some syntactic substitutions would also be required. The user-interface of ORT would require change and this emulation would have to be installed before ORT could be adapted to more conventional libraries.

- The approach makes little attempt to interpret the information stored in the library. If ORT were to be extended to directly aid in the process of re-engineering existing software systems, it seems reasonable to expect the addition of some analysis functionality as well as some higher-level representation for discussing a software system made up of classes stored in the library.

8.5 Benefits of the ORT Approach

In closing, there are a number of benefits ORT provides to those wishing to take software reuse seriously. The primary benefit of ORT is that it serves as a tool to aid in producing designs which maximally reuse existing classes. The set of benefits described and demonstrated in Chapter 6 are also significant:

1. Assistance in Assessing Reusability – Information in the library allows ORT users to quickly assess characteristics relating to the reusability of a class.

2. Querying Stimulates Design Improvements – Ideas related to a desired concept can be readily found in query results and may stimulate the user to improve the design.

3. Non-local Browsing – The flexibility of the search capabilities provided by ORT increase the probability of finding relevant components.

4. Identifying Possible Design Modifications – The knowledge held by the ORT library can implicitly suggest standard solutions to well thought-through problems and can thus improve the set of classes used to construct a system.

Further, there are a number of benefits that are considered second-order or difficult to demonstrate clearly. They are left for the reader to as a suggestion that ORT may yield more promise as a practical tool for modern software development environments.

- It may be human nature to resist exposing design ideas to others until they seem well thought-out and complete. By this time, it is often too late to "submit" to design changes to maximally reuse existing software. However, a designer may be more effective if he or she can bounce possibly "half-baked" ideas off of ORT and get feedback with respect to reuse potential while not "loosing face" to a colleague.

- ORT can facilitate learning about system architecture and how to build good systems by providing well-documented examples in its library.

- ORT has the capability of becoming a medium in which software engineers can communicate and exchange information. This communication is focused tightly on the individual products of the development process and directed towards maximizing their quality and minimizing duplicated effort.

- ORT provides the ability to collect and organize metrics concerning the design, documentation, coding, testing, and maintenance activities of a class. These metrics can be computed and stored in a more structured and automated way than current practice allows.

- The use of ORT does not prohibit the user from also using other important software development tools. ORT allows the orthogonal integration of a separate source configuration management system and a separate defect-tracking system, because a File entity in the model can any file in the same virtual file-system as ORT.

- ORT provides a more structured environment to manage the documentation of classes, including its storage, revision control, perusal, display, and printing.

- The task of managing dependencies within different versions of sets of classes, a cumbersome task in a paper documentation approach, is assisted by ORT.

- The usage history of a Class Description can serve as a collection of examples of how the associated class is used, which can be important to programmers who learn well by example.

- A usage history can also provide the names of programmers that should be contacted in the event of an update due to a "bug-fix" or enhancement. This provides an incentive to register reuse efforts in the library, as this will allow eventual notification of relevant changes.

Bibliography

[1] American National Standards Institute. *Database Language SQL.* ANSI X3.135-1986.

[2] C. Arapis and G. Kappel. Organizing Objects in an Object Software Base. Technical report, Universite de Geneve, June 1988.

[3] Nabiha Azar and Entienne Pichat. Translation of an Extended Entity-Relationship Model into the Universal Relation with Inclusions Formalism. In Stefano Spaccapietra, editor, *Entity-Relationship Approach.* Elsevier Science Publishers B. V. (North-Holland), 1987.

[4] Brian Beach. Object Protocols for Software Reuse. In *Proc. HP Software Engineering Productivity Conf.*, pages 2–2 – 2–14, May 1987.

[5] Ted Biggerstaff and Charles Richter. Reusability Framework, Assessment and Directions. *IEEE Software*, 4(2):41–49, March 1987.

[6] Barry W. Boehm and Philip N. Papaccio. Understanding and Controlling Software Costs. *IEEE Trans. on Software Engineering*, 14(10):1462–77, October 1988.

[7] Marco A. Casanova. Designing Entity-Relationship Schemes for Computer Information Systems. In P. P. Chen, editor, *Entity-Relationship Approach to Software Engineering.* Elsevier Science Publishers B. V. (North-Holland), 1983.

[8] Peter Chen. The Entity Realationship Approach. *BYTE*, April 1988.

[9] T. C. Chiang and G. R. Rose. Design and Implementation of An E-R Data Base Management System (DBM-2). In P. P. Chen, editor, *Entity-Relationship Approach to Information Modeling and Analysis.* Elsevier Science Publishers B. V. (North-Holland), 1983.

[10] Jeff Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, pages 17–41, September 1987.

[11] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, April 1987.

[12] C. J. Date. *A Guide to the SQL Standard.* Addison Wesley, 1987.

[13] T. DeMarco and T. Lister. *Controlling Software Projects: Management, Measurement, and Evaluation.* Atlantic Systems Guild Inc., 1984.

[14] Bruce Burton et al. The Reusable Software Library. *IEEE Software*, pages 25–33, July 1987.

[15] Fishman et al. Iris: An Object-Oriented Database Management System. *ACM Trans. on Offic Information Systems*, 5(1):48–69, January 1987.

[16] Hiroyuki Tarumi et al. A Programming Environment Supporting Reuse of Object-Oriented Software. In *Proc. 10th Intl. Conf. on Software Engineering*, pages 265 – 273, April 1988.

[17] Tim O'Reilly et al. *X Window System User's Guide.* O'Reilly and Associates, 1988.

[18] Y. Matsumoto et al. SWB System: A Software Factory. In H. Hunke, editor, *Software Engineering Environments*, pages 305–318, 1981.

[19] Steve Fiedler. Object-Oriented Unit Testing. In *Proc. HP Software Engineering Productivity Conf.*, pages 297–310, August 1988.

[20] W.B. Frakes and B.A. Nejmeh. Software Reuse Through Information Retrieval. In *COMPCON Spring 87*, pages 380–384, February 1987.

[21] W. L. Frank. What Limits To Software Gains. *Computerworld*, pages 65–70, May 4 1984.

[22] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[23] Keith E. Gorlen. Object-Oriented Program Support: OOPS Version 1 Ref. Manual. Draft Version, May 1986.

[24] Beth Farrell Hewlett-Packard, November 1988. Personal Communication.

[25] Sushil Jajodia and Peter Ng. On Representation of Relational Structures by Entity-Relationship Diagrams. In P. P. Chen, editor, *Entity-Relationship Approach to Software Engineering.* Elsevier Science Publishers B. V. (North-Holland), 1983.

[26] T. Capers Jones. Reusability in Programming: A Survey of the State of the Art. *IEEE Trans. on Software Engineering*, SE-10(5):488–494, September 1984.

[27] Gail Kaiser and Peter Feiler. An Architecture for Intelligent Assistance in Software Development. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 180 – 188, April 1987.

[28] Gail Kaiser and David Garlan. Melding Software Systems from Reusable Building Blocks. *IEEE Software*, pages 17–24, July 1987.

[29] R. Lanergan and B. A. Poynton. Reusable Code — The application development technique of the future. In *Proc. Joint SHARE/GUIDE/IBM Appl. Develop. Symp.*, pages 127–136, October 1979.

[30] Karl Lieberherr and Arthur Riel. Demeter: A CASE Study of Software Growth Through Parameterized Classes. *Journal of Object-Oriented Programming*, pages 8–22, August/September 1988.

[31] M. D. McIlroy. Mass-Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, January 1969. NATO Sci. Committee.

[32] Bertrand Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, 4(2):50–64, March 1987.

[33] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, 1988.

[34] Ruben Prieto-Diaz and Peter Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

[35] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Program Synthesis Scenario. Technical Report 933, MIT A.I. Laboratory, November 1986.

[36] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[37] Will Tracz. Software Reuse Myths. In *ACM SIGSOFT Software Engineering Notes Vol. 13 No. 1*, pages 17–21, January 1988.